AFRL-IF-RS-TR-2004-127
Final Technical Report
May 2004

FLEXIBLE AND SCALABLE METHODS FOR MULTI-AGENT DISTRIBUTED RESOURCE ALLOCATIONS BY EXPLOITING PHASE TRANSITIONS

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K278

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-127 has been reviewed and is approved for publication

APPROVED: /s/
ROBERT J. PARAGI
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 2004	**FINAL** **May 00 – Sep 03**

4. TITLE AND SUBTITLE

FLEXIBLE AND SCALABLE METHODS FOR MULTI-AGENT DISTRIBUTED RESOURCE ALLOCATIONS BY EXPLOITING PHASE TRANSITIONS

5. FUNDING NUMBERS

G - F30602-00-2-0531
PE - 62301E
PR - ANTS
TA - 00
WU - 02

6. AUTHOR(S)

Weixiong Zhang

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Southern California
Department of Contracts and Grants
837 West Downey Way, Room 325
Los Angeles CA 90089-1147

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRLIF-RS-TR-2004-127

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Robert Paragi/IFTB/(315) 330-3547 Robert.Paragi@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This report summarizes the main findings and results on five independent yet closely related research topics, which were motivated by some difficult distributed constraint problems from Autonomous Negotiating Teams (ANTs) domains: (1) distributed constraint problem solving in sensor networks and low overhead distributed constraint algorithms, (2) analysis of negotiation protocols as distributed search, (3) phase transitions and backbones of the Traveling Salesman Problem, (4) configuration space analysis of Boolean satisfiability and backbone-guided local search, and (5) improved integer local search algorithms for complex scheduling problems. Our results provide deep understanding of the difficulty and complexity of distributed constraint problems in sensor networks and distributed environments, and produce effective and efficient methods, algorithms and software for these difficult problems.

14. SUBJECT TERMS

autonomous negotiation, autonomous agents, software agent system complexity and dynamics, intelligent agents, computational phase transitions

15. NUMBER OF PAGES 120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

Acknowledgement

Thanks to Stephen Fitzpatrick for many discussions related to distributed stochastic search, and to Alejandro Bugacov for many helpful discussions on crew scheduling problems. In particular, the following researchers and graduate students participated in this research, Sharlee Climer, Zidong Deng, Moshe Looks, Guandong Wang, Lars Wittenburg, Zhao Xing, Weihong Zhang, Weixiong Zhang and Xiaotong Zhang.

1 Summary and Introduction

We have touched upon in this research many aspects of distributed constraint problems in different distributed environments and settings. Our results provided insights to some of the intrinsic issues of problem solving in distributed environments. The final products of this research include new understanding of phase transitions and backbones of combinatorial optimization problems, such as the Traveling Salesman Problem, new algorithms for complex and distributed constraint satisfaction and optimization problems, and corresponding software.

Our research has been mainly focused on five topics. We briefly summarize the main findings and results in the following five paragraphs. Each of these topics will be discussed in great detail in a separate section.

The first topic is distributed constraint solving and low overhead distributed constraint algorithms. We specifically consider constraint problem solving, e.g., distributed scheduling, in distributed sensor networks. We study such distributed algorithms that have low overhead on computation and communication. We first cast such problems as distributed constraint satisfaction problems (DisCSP) and distributed constraint optimization problems (DisCOP) and model them as distributed multiple coloring. To cope with limited resources and restricted real-time requirement, we consider distributed algorithms that have low overhead on resource consumption and high anytime performance. To meet these requirements, we study two existing DisCSP algorithms, distributed stochastic search algorithm (DSA) and distributed breakout algorithm (DBA), for solving DisCOPs and the distributed scheduling problems. We experimentally show that DSA has a phase-transition or threshold behavior, in that its solution quality degenerates abruptly and dramatically when the degree of parallel executions of distributed agents increases beyond some critical value. We also consider the completeness and complexity of DBA for graph coloring. We show that DBA is complete on acyclic graphs and is able to color an acyclic graph of n nodes in $O(n^2)$ steps. However, on a cyclic graph, DBA may never terminate. To improve DBA's performance on coloring cyclic graphs, we propose two stochastic variations. Finally, we directly compare DSA and DBA for solving distributed multiple coloring and distributed scheduling problems in sensor networks. The results show that DSA is superior to DBA when controlled properly, having better or competitive solution quality and significantly lower communication cost than DBA. Therefore, DSA is the algorithm of choice for our distributed scheduling problems.

The second topic is negotiation as distributed search. Negotiation is one of the main mechanisms for coordination and cooperation in distributed environments. However, most negotiation protocols are complex and their features are difficult to characterize. We propose a general experimental approach to analyzing negotiation strategies using distributed search. In this approach we first formulate the problems that negotiation protocols intend to solve as distributed constraint satisfaction/optimization problems, and then capture the negotiation protocols as distributed search algorithms. By analyzing the derived search algorithms, we can characterize many important properties of the negotiation protocols. In this paper, we are particularly interested in the properties of a newly developed negotiation protocol, which is motivated by distributed sensor network applications, including its completeness, complexity, convergence rate, and

scalability. Although the idea of viewing negotiation as distributed search is not completely new, in this research we not only view negotiation as distributed search, but directly apply a search algorithm to reveal the essential features of a negotiation protocol and analyze its performance.

The third topic is phase transitions and backbones of combinatorial optimization problems, the Traveling Salesman Problem in particular. In recent years, phase transitions have been successfully used to analyze combinatorial optimization problems, characterize their typical-case features and locate the hardest problem instances. In this research, we empirically study phase transitions of the asymmetric Traveling Salesman, an NP-hard combinatorial optimization problem that has many real-world applications. Using random instances of up to 1,500 cities, we show that many properties of the problem, including the optimal tour cost and backbone, experience sharp transitions as the precision of intercity distances increases across a critical value. Our experimental results on the costs of the Asymmetric Traveling Salesman (ATSP) tours and assignment problem support the theoretical result that the asymptotic cost of assignment problem is $\pi^2/6$. In addition, we show that the average computational cost of the well-known branch-and-bound subtour elimination algorithm for the problem also exhibits a threshold behavior, transitioning from easy to difficult as the distance precision increases. These results answer positively an open question regarding the existence of phase transitions in the Traveling Salesman, and provide guidance on how difficult ATSP problem instances should be generated.

The fourth topic is a novel metaheuristic for local search in solving Boolean satisfiability. Many constraint problem can be captured as Boolean satisfiability (SAT) and maximum satisfiability (Max-SAT), which are difficult combinatorial problems that have many important real-world applications. In this research, we investigate the configuration landscapes of local minima reached by the WalkSAT local search algorithm, one of the most effective algorithms for SAT. A configuration landscape of a set of local minima is their distribution in terms of quality and structural differences relative to an optimal or a reference solution. Our experimental results show that local minima from WalkSAT form large clusters, and their configuration landscapes constitute big valleys, in that high quality local minima typically share large partial structures with optimal solutions. Inspired by this insight into WalkSAT and the previous research on phase transitions and backbones of combinatorial problems, we propose and develop a novel method that exploits the configuration landscapes of such local minima. The new method can be embedded in a local search algorithm, such as WalkSAT, to improve its performance. On large problem instances from a SAT library (SATLIB), the backbone guided WalkSAT algorithm finds satisfiable solutions to SAT problems with higher probabilities than WalkSAT, and obtains better solutions on Max-SAT problems, improving solution quality by 20% on average.

The last topic is a set of effective methods for complex scheduling problems. We consider complex scheduling problems that can be captured as optimization under hard and soft constraints. The objective of such an optimization problem is to satisfy as many hard constraints as possible and meanwhile to minimize a penalty function of unsatisfied soft constraints. We present an efficient local search algorithm for these problems which improves upon WalkSAT(oip), a WalkSAT-based local search algorithm for overconstrained problems represented in integer programs. We introduce three techniques to the WSAT(oip) algorithm to extend its capability and improve its performance: backbone guided biased moves to drive the search to

the regions in search space where high-quality and optimal solutions reside; sampling-based aspiration search to reduce search cost and make anytime solutions available over the course of the search; and dynamic parameter tuning to dynamically adjust the key parameters of the algorithm to make it robust and flexible for various applications. Our experimental results on large-scale crew scheduling, basketball tournament scheduling and progressive party scheduling show that the new improved algorithm can find better solutions with less computation than WSAP(oip).

Most of the research described in this report has been published in many conference and journal papers, which include those of [100, 110, 111, 115, 116, 117, 118, 119, 120].

The rest of the report will describe these topics and results in detail. As these topics are relatively independent, we draw conclusions for each of them, at the end of their corresponding sections.

2 Low Overhead Distributed Constraint Algorithms

In recent years, various micro-electro-mechanical systems (MEMS) devices, such as sensors and actuators with some information processing capabilities embedded within, have been developed and deployed in many real-world applications [85, 93]. For example, one application that we have been involved with is detecting and tracking mobile objects in dynamic and real-time environments, where distributed sensors must cooperatively monitor an area to detect new objects and some of the sensors have to sense and measure at the same time in order to triangulate a target. Such distributed sensor environments were specific research platforms in the Autonomous Negotiating Teams (ANTs) domains. In such distributed applications, there are logical restrictions among the actions of the sensors and actuators and restrictions on available computational and communication resources as well as restrictions on energy consumption.

Multi-agent system technology can play a critical role in developing large-scale networked, embedded systems using such smart devices, by providing frameworks for building and analyzing such systems. Due to the real-time nature of many applications and limited computational resources on the devices, e.g., slow CPUs and small memories, the key to large-scale, real-time MEMS is the mechanism that the smart devices (or agents) use to make viable distributed decisions in restricted time with limited computational resources. Therefore, the methods for distributed constraint problem solving are also important tools for such real-time decision making in distributed environments.

In contrast to the current research on sensor networks, which has largely focused on building ad hoc, mobile communication networks (e.g., [40, 121]), this line of research on applying multi-agent system technology and constraint problem-solving techniques has a paramount importance to the overall success of embedded sensor networks by addressing the key issue of resource allocation at the application level.

We have been developing large-scale sensor networks controlled by distributed multi-agent systems for the applications of mobile object detection and tracking problem and the vibration dampening in the avionics domain. We place an agent on top of a sensor, which runs on a battery and has, among other things, a slow CPU, a small memory and a wireless communication unit. The sensors/agents need to be programmed in such a way that they can collaboratively carry out a task, which no single sensor/agent can do on its own. In this research, we are particularly interested in *low-overhead methods* for solving DisCSPs and DisCOPs. By low-overhead methods we mean those in which limited communication among agents is needed and agents make their decisions without global knowledge of the system and without knowing the actions of the other agents.

In this research, we study distributed stochastic algorithm (DSA) [25, 27, 67] and distributed breakout algorithm (DBA) [79, 103, 105], two existing algorithms for DisCSP, for the purpose of solving our distributed scheduling problems in sensor networks. In other words, we apply and extend these algorithms to solving DisCOP, which is much more difficult than DisCSP.

Note that DSA and DBA are incomplete in that they do not guarantee to find satisfying solutions to DisCSPs and thus may not find optimal solutions to DisCOPs either. Nevertheless, we choose these algorithms for two reasons. First, compared to complete distributed algorithms, such as those of [103, 104], DSA and DBA have low computational and communication cost. An agent in these algorithms only

needs information local to itself and information of neighboring agents connected through the underlying constraint graphs. Therefore, the computational cost is low. These algorithms also do not have to record many previous states visited, so that memory requirement is low as well. Second, as we will see later in this section, these algorithms have good anytime performance, finding good approximate solutions quickly. Even if they are not guaranteed the optimality of solutions that they provide, such heuristic algorithms are sufficient for finding good approximate solutions in real-time environments where the targeting objectives may also change over time, making optimal solutions less important.

In addition to the above important characteristics to serve the needs of some applications in sensor networks, DSA and DBA also have unique features in resolving conflicts among distributed agents. DBA introduces priorities among agents' actions for conflict resolution, while DSA uses randomness to possibly avoid conflicts. Moreover, as shown in this section, DSA has a threshold behavior, in which solution quality degrades abruptly and dramatically as the randomness for conflict resolutions increases beyond a critical value, and DSA becomes complete and has a low polynomial time complexity for distributed multiple coloring problems when the underlying constraint graphs are acyclic. However, on other constraint graph structures, including the constraint graphs constructed from our motivating applications, DSA is superior to DBA, generally finding better solutions with less time. Overall, our experimental results show that DSA is a good choice of algorithm for our sensor network applications.

This section is organized as follows. A detailed account of the scheduling problems is given in Section 2.1. We formulate the problems as weighted multi-coloring problems in Section 2.2. We then describe DSA and analyze its threshold behavior in Section 2.3. We consider DBA and its completeness and complexity for graph coloring in Section 2.4. We also discuss introducing randomness to improve DBA's performance. We compare the performance of DSA and DBA on distributed multiple graph coloring and our distributed scheduling problem of finding the shortest scan schedule for mobile object detection in Section 2.5. We discuss related work in Section 2.6, and finally conclude in Section 2.7.

Early results of this section have appeared in [111, 116, 118, 120].

2.1 Distributed Constraint Satisfaction and Optimization in Sensor Networks

To set the stage for our investigation of distributed algorithms for distributed constraint problems, we first describe our application problems in sensor networks.

Detecting and tracking mobile objects in large open environments is an important topic that has many real applications in different domains, such as avionics, surveillance and robot navigation. We are developing such an object detecting and tracking system using MEMS sensors, each of which operates under restricted energy sources, i.e., batteries, and has a small amount of memory and restricted computational power. In a typical application of our system, a collection of small Doppler sensors are scattered in an open area to detect possible foreign objects moving into the region. Each sensor can scan and detect an object within a fixed radius. However, the overall detecting area of a sensor is divided into three equal sectors, and the sensor can only operate in one sector at any given time. The sensors can communicate with one another through radio communication. The radio channel is not reliable, however, in that a signal may

get lost due to, for instance, a collision of signals from multiple sensors, or distorted due to environment noises. Moreover, switching from one scanning sector to another sector and sending and receiving radio signals take time and energy. To save energy, a sensor may turn itself off occasionally or periodically if doing so does not reduce system performance.

The overall system operates to achieve two conflicting goals, to detect foreign objects as many and as quickly as possible and meanwhile to preserve energy as much as possible so as to prolong the system's lifetime. The overall problem is thus to find a distributed scan schedule to optimize an objective function that balances the above two conflicting goals.

One of our goals here is to develop a scalable sensor network for object detection in that the system response time for detecting a new object does not degenerate when the system size increases. To this end, we place an agent on top of each sensor and make the agent as autonomous as possible. This means that an agent will not rely on information of the whole system, rather the local information including its neighboring sensors or agents. These agents also try to avoid unnecessary communication so as to minimize system response time. Therefore, complex coordination and reasoning methods are not suitable.

We set forth to apply simple, low-overhead methods for solving this distributed scheduling problem. We assume that the sensors are stationary, in that they will not move after being placed in an operation environment. We consider two distributed algorithms, distributed breakout and distributed stochastic search, that were developed earlier to solve distributed constraint satisfaction problems. Our objectives are twofold. First, we want to understand how these two algorithms compare with each other on a real-world application such as our scan scheduling problem. Second, we want to identify the right application condition and parameters of the algorithm that we will use based on the evaluation.

2.2 Constraint Models in Distributed Multiple Coloring

Our first step to address the distributed scheduling problems is to model them as distributed constraint satisfaction and optimization problems. Indeed, the core problems in our applications discussed in the previous section can be captured as distributed multi-coloring problems.

In the mobile object detection problem, a sensor needs to scan its three sectors as often as possible. However, to save energy, two neighboring sensors should try to avoid scanning a common area covered by two overlapping sensor sectors at the same time since one sensor's sensing of an area is sufficient to detect possible objects in the area. The objective is to find a sequence of sensing actions so that each sensor sector can be scanned at least once within a minimal period of time. In other words, the objective is a cyclic scan schedule in which each sector is scanned at least once with the constraint of minimizing the energy usage.

As all agents execute the same procedure, scanning actions on different sensors take approximately the same amount of time. Assume that the sensors are synchronized[1], an assumption that can be lifted by a synchronization method[2] [94]. Therefore, the problem can be viewed as coloring a weighted graph so

[1]The algorithms we will consider shortly, DSA and DBA, are synchronous. It has also been observed that under certain conditions asynchronized actions may actually improve the performance of a distributed algorithm [26].

[2]Indeed, DBA and its synchronization mechanism have been implemented and tested on a real-time middleware using Object Request Broker framework

that the total weight of violated constraints is minimized. Here, a node corresponds to a sensor or agent, a link between two nodes represents the constraint of a shared region between two agents, and the weight captures the area of a common region. The larger a shared region is, the more energy will be wasted if two sensors scan the shared region at the same. Moreover, each color corresponds to a time slot in which a sector is scanned and the total number of colors represents the scanning cycle length or the number of time slots required to scan each of the sectors of all sensors. A node must have at least one color so that the corresponding sector is scanned at least once, and may also have multiple colors so that the sector may be sensed multiple times to increase system's object detection rate.

We further consider a sensor as an hypernode of three nodes, each of which corresponds to a sector of the sensor. Within an hypernode, two nodes cannot share a color since two sectors cannot be scanned at the same time. In other words, the colors assigned to the nodes within an hypernode are mutually exclusive. This is a hard constraint that cannot be violated. Furthermore, not all available colors must be used as a sensor may sometimes turn itself off. A node within a hypernode may also be constrained by a node of another hypernode if the two corresponding sectors share a common region. This constraint may be violated, resulting in wasted energy. Overall, we are looking for such a coloring that satisfies all hard constraints within hypernodes and minimizes the weights of violated constraints among hypernodes so as to reduce the overall conflicts of scanning common regions among overlapping sectors.

We can model the distributed scan scheduling problem as follows. Let $G(V, E)$ be an undirected graph with a set of hypernodes V and a set of edges E. Each hypernode $v \in V$, which represents a sensor, consists of k nodes, $v_1, v_2, \cdots, v_k$, which model k sectors of the sensor. (In our application, we have $k = 3$.) An edge between nodes $u_i \in u$ and $v_j \in v$ is denoted as $(u_i, v_j) \in E$, and u_i and v_j are called neighbors. Two hypernodes are neighbors if any pair of their nodes are neighbors. The weight of an edge (u_i, v_j), denoted as $w(u_i, v_j)$, is the area of a common region covered by sectors of sensors u and v. Every hypernode is given a total T *available colors* that can be used and a sensor activation ratio $\alpha \leq 1$. $|T|$ corresponds to a schedule length. α captures the frequency of how often a sensor should be active, and $1 - \alpha$ is the frequency determining how often a sensor should turn itself off to save energy. Given the total available colors T, the weighted multi-coloring problem is to color the nodes following these criteria. First, exactly $\lceil \alpha T \rceil$ colors are used within a hypernode. This means that a node may have more than one color if the available colors are more than the nodes in a hypernode, i.e., $\lceil \alpha T \rceil > k$. Second, every node must have at least one color and no two nodes within a hypernode can share a color. Third, minimizing conflicts between pairs of nodes in different hypernodes, so the total weight of violations going across hypernodes is minimized. When no conflict is allowed or a minimal level of allowed conflicts is specified, the overall problem is to find the minimal number of allowed colors T and such a schedule satisfying the criteria.

The distortion and damage detection problem can also be captured by a constraint model. Scheduling the signaling activities of the ping nodes can be formulated as a distributed graph coloring problem. A color here corresponds to a time slot in which a ping node sends out signals. The number of colors is therefore the length in time units of a schedule. The problem is to find a shortest schedule such that the

pinging signals do not interfere with one another in order to increase damage detection response time and reduce the amount of wasted energy. The problem is equivalent to finding the chromatic number of a given constraint graph, which corresponds to the minimal worst-case response time and a coloring of the graph within the overall system response time.

In short, this damage detection problem is a distributed constraint satisfaction/optimization problem with variables and constraints distributed among agents. Collectively the agents find a solution to minimize an objective function, which is the number of violated constraints.

2.3 Distributed Stochastic Search

In this section, we study distributed stochastic search algorithm. In fact, this is not a single algorithm, but a family of distributed algorithms, as we will see shortly.

2.3.1 The algorithm

Distributed stochastic algorithm (DSA) is uniform [94], in that all processes are equal and have no identities to distinguish one another. It is also synchronous in principle [94], in that all processes proceed in synchronized steps and in each step it sends and receives (zero or more) messages and then performs local computations, i.e., changing local state. Note that synchronization in DSA is not crucial since it can be achieved by a synchronization mechanism [94].

The idea of DSA and its variations is simple [25, 27, 67, 83]. After an initial step in which the agents pick random values for their variables, they go through a sequence of steps until a termination condition is met. In each step, an agent sends its current state information, i.e., its variable value in our case, to its neighboring agents if it changed its value in the previous step, and receives the state information from the neighbors. It then decides, often stochastically, to keep its current value or change to a new one. The objective for value changing is to possibly reduce violated constraints. A sketch of DSA is in Algorithm 1.

The most critical step of DSA is for an agent to decide the next value, based on its current state and its believed states of the neighboring agents. If the agent cannot find a new value to improve its current state, it will not change its current value. If there exists such a value that improves or maintains state quality, the agent may or may not change to the new value based on a stochastic scheme.

Table 1 lists five possible strategies for value change, leading to five variations of the DSA algorithm. In DSA-A, an agent will change its value only when the state quality can be improved. DSA-B is the same as DSA-A except that an agent may also change its value if there is a violated constraint and changing its value will not degrade state quality. DSA-B is expected to have a better performance than DSA-A since by reacting stochastically when the current state cannot be improved directly ($\Delta = 0$ and there exists a conflict), the violated constraint may be satisfied in the next step by the value change at one of the agents involved in the constraint. Thus, DSA-B will change value more often and has a higher degree of parallel actions than DSA-A.

Furthermore, DSA-C is more aggressive than DSA-B, changing value even if the state is at a local minima where there exist no conflict but another value leading to a state of the same quality as the current

Algorithm 1 Sketch of DSA, executed by all agents.

```
Randomly choose a value
while (no termination condition is met) do
  if (a new value is assigned) then
    send the new value to neighbors
  end if
  collect neighbors' new values, if any
  select and assign the next value (See Table 1)
end while
```

Algo.	$\Delta > 0$	C, $\Delta = 0$	no C, $\Delta = 0$
DSA-A	v with p	-	-
DSA-B	v with p	v with p	-
DSA-C	v with p	v with p	v with p
DSA-D	v	v with p	-
DSA-E	v	v with p	v with p

Table 1: Next value selection in DSAs. Here C stands for conflict, Δ is the best possible conflict reduction between two steps, v the value giving Δ, and p a probability to change the current value, which represents the degree of parallel executions, and "-" means no value change. Notice that when $\Delta > 0$ there must be a conflict.

one. An agent in DSA-C may move to such an equal-quality value in the next step. It is hoped that by moving to another value, an agent gives up its current value that may block any of its neighbors to move to a better state. Therefore, the overall quality of the algorithm may improve by introducing this equal-quality action at a single node. The actual effects of this move remain to be examined, which is one of the objectives of this research.

Parallel to DSA-B and DSA-C, we have two more aggressive variations. DSA-D (DSA-E) extends DSA-B (DSA-C) by allowing an agent to move, deterministically, to a new value as long as it can improve the current state ($\Delta > 0$). These variations make an agent more greedily self centered in that whenever there is a good move, it will take it.

Notice that the level of activities at an agent increase from DSA-A, to DSA-B and to DSA-C, and from DSA-D to DSA-E. The level of activities also reflects the degree of parallel executions among neighboring processes. When the level of local activities is high, so is the degree of parallel executions.

To change the degree of parallel executions, an agent may switch to a different DSA algorithm, or change the probability p that controls the likelihood of updating its value if the agent attempts to do so. This probability controls the level of activities at individual agents and the degree of parallel executions among neighboring processes. One major objective of this research is to investigate the effects of this control parameter on the performance of DSA algorithms.

The termination conditions and methods to detect them are complex issues of their own. We will adopt a termination detection algorithm [94] in a later stage. In our current implementation, we terminate DSAs after a fixed number of steps. This simple determination method serves the basic needs of the current research, i.e., experimentally investigating the behavior and performance of these algorithms, one of the

main focuses of this section.

2.3.2 Phase Transitions

DSAs are stochastic, in that they may behave differently even if all conditions are equal. We are interested in their typical or statistical behavior at an equilibrium state when the behavior of the algorithms does not seem to change dramatically from one step to the next. We are specifically interested in the relationship between the degree of parallel executions, controlled by the probability p (cf. Table 1), and the performance of the algorithms, including their solution quality and communication costs.

It turns out that the performance of DSAs may experience phase transitions on some constraint structures when the degree of parallelism increases. Phase transitions refer to a phenomenon of a system in which some global properties change rapidly and dramatically when a control or order parameter goes across a critical value [14, 44]. A simple example of a phase transition is water changing from liquid to ice when the temperature drops below the freezing point. For the problem of interest here, the system property is DSAs performance (solution quality and communication cost) and the order parameter is the probability p that controls the degree of parallel executions of the agents.

Solution quality: We experimentally investigate DSAs' phase-transition behavior on grids, random graphs and trees. In our experiments, we used different networks, including grids, which appear in our motivating application, and graphs and trees. We considered graph coloring problems, by varying the connectivity of the structures and the number of colors used, we are able to generate underconstrained, critically constrained and overconstrained problem instances. In the following discussions, we will focus on grid and graph structures.

Starting from random initial colorings, we let the algorithms run for a large number of steps, to the point where they seem to reach an equilibrium, i.e., the overall coloring quality does not change significantly from one step to the next. We then measure the solution quality, in terms of the number of constraints violated. In our experiments, we measure the performance at 1,000 steps; longer executions, such as 5,000 and 10,000 steps, exhibit almost the same results.

The distributed algorithms were simulated on one machine using a discrete event simulation method [103]. In this method, an agent maintains a step counter, equivalent to a simulated clock. The counter is increased by one after the agent has executed one step of computation, in which it sends its state information, if necessary, receives neighbors' messages, and carries out local computation. The overall solution quality is measured, at a particular time point, by the total number of constraints violated, and the communication cost is measured by the total number of messages sent.

We varied the degree of parallel executions, the probability p in Table 1, and examined the quality of the colorings that DSAs can provide. The solution quality indeed exhibits phase-transition behavior on grid and graph structures as the degree of parallelism increases.

Grids We generate grids of various sizes, including 20×20, 40×40 and 60×60 grids, and use different number of colors, ranging from two to eight. In order to study how DSAs will scale up to large problems, we simulate infinitely large grids. We remove the grid boundaries by connecting the nodes on the top to

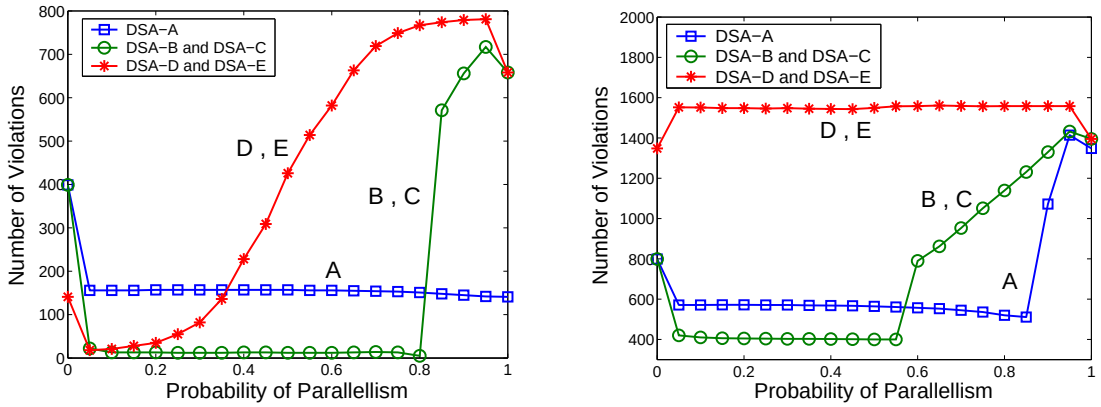


Figure 1: Solution quality phase transitions on 2-coloring grids; $k = 4$ (left) and $k = 8$ (right).

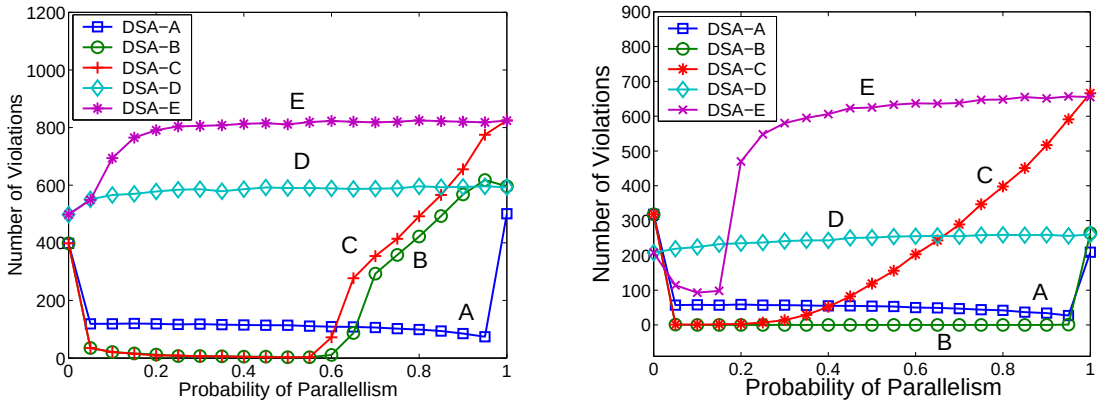


Figure 2: Solution quality phase transitions on grids; $k = 8$ using 4 colors (left) and 5 colors (right).

those on the bottom as well as the nodes on the left to those on the right of the grids. We also change the degree of constrainedness by changing the number of neighbors that a node may have. For example, on a degree $k = 4$ grid, each node has four neighbors, one each to the top, the bottom, the left and the right. Similarly, on a degree $k = 8$ grid, each node has eight neighbors, one each to the top left, top right, bottom left and bottom right in addition to the four neighbors in a $k = 4$ grid.

Figure 1 shows the total numbers of constraint violations after 1,00 steps of the algorithms using two colors on 20×20 grids with $k = 4$ (Figure 1(left)) and $k = 8$ (Figure 1(right)). Each data point of the figure is averaged over 1,000 random initial colorings. Note that the results from larger grids, such as 40×40 grids, follow almost identical patterns as in Figure 1.

The figures show that DSAs' phase-transition behavior is controlled by the degree of parallelism, except DSA-A on grids with $k = 4$. The transitions are typically very sharp. For example, as Figure 1(left) shows, the solution quality of DSA-B and DSA-C decreases abruptly and dramatically when the probability p increases above 0.8. More importantly and surprisingly, after the transition, the solution quality is even worse than a random coloring. The average solution quality of random colorings corresponds to the point $p = 0$ on the DSA-B and DSA-C curves in the figure. This indicates that the degree of parallel exe-

cutions should be controlled under a certain level in order for the algorithms to have a good performance. Furthermore, the transitions start earlier for DSA-D and DSA-E. Although DSA-A, the most conservative algorithm, does not show phase transitions on grids of $k = 4$, its average solution quality is much worse than that of DSA-B, because it may be easily trapped in local minima.

The degree of parallelism and the constrainedness of the underlying network structures also interplay. Grids with $k = 4$ are 2-colorable while grids with $k = 8$ are not, and are thus overconstrained. The results shown in Figure 1 indicate that the phase transitions appear sooner on overconstrained problems than on underconstrained problems. Even the most conservative DSA-A also experiences a phase transition when $k = 8$. The most aggressive ones, DSA-D and DSA-E, always performs worse than a random coloring on this overconstrained grid.

A coloring problem becomes easier if more colors are used, since it is less constrained to find a satisfying color in the next step. However, the phase-transition behavior persists even when the number of colors increases. Figure 2 shows the results on grids with $k = 8$ using 4 and 5 colors. Notice that the curves in the 4-color figure and the curves for 3 colors (not shown here) follow similar patterns as in the case for 2 colors in Figure 1(right).

Graphs The phase transitions of DSAs persist on graphs as well, and follow similar patterns as in the grid cases. We conducted experiments on random graphs. We generate graphs with 400 and 800 nodes and average node connectivity equal to $k = 4$ and $k = 8$. A graph is generated by adding edges to randomly selected pairs of nodes. These two types of graphs are used to make a correspondence to the grid structures of $k = 4$ and $k = 8$ mentioned before. We also generated random trees with depth four and average branching factors $k = 4$ and $k = 8$.

Figure 3 shows the results on graphs with $k = 4$ and $k = 8$ using 2 colors, and Figure 4 the results on graphs with $k = 8$ using 4 and 5 colors. Each data point is an average of 1,000 random instances. The solution quality is also measured after 1,000 steps of executions. As all the figures show, the phase transitions on random graphs have similar patterns as those on grids. Therefore, the discussions on the grids apply in principle to random graphs. We need to mention that on graphs, the most aggressive algorithms, DSA-D and DSA-E, do not perform very well under all degrees of parallel executions. This, combined with the results on grids, leads to the conclusion that DSA-D and DSA-E should not be used.

Trees There is no phase transition observed on random trees in our tests. All DSAs perform poorly on 2-coloring, in comparison with their performance on grids and graphs. This seems to be counterintuitive since trees have the simplest structures among all these network structures. One explanation is that DSAs may be easily trapped into local minima. Since trees are always 2-colorable, we are able to easily create local minima in which none of DSAs can escape.

Communication cost: We have so far focused on DSAs' solution quality without paying any attention to their communication costs. Communication in a sensor network has an inherited delay and could be unreliable in many situations. Therefore, communication cost of a distributed algorithm is an integral part of its overall performance. It is desirable to keep communication cost as low as possible.

In fact, the communication cost of a DSA algorithm goes hand-in-hand with its solution quality. Recall

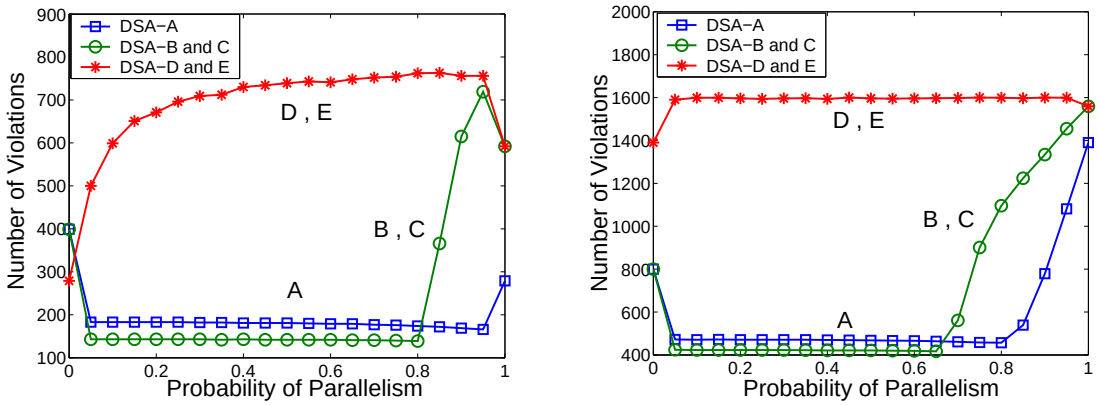


Figure 3: Solution quality phase transitions on 2-coloring graphs; $k = 4$ (left) and $k = 8$ (right).

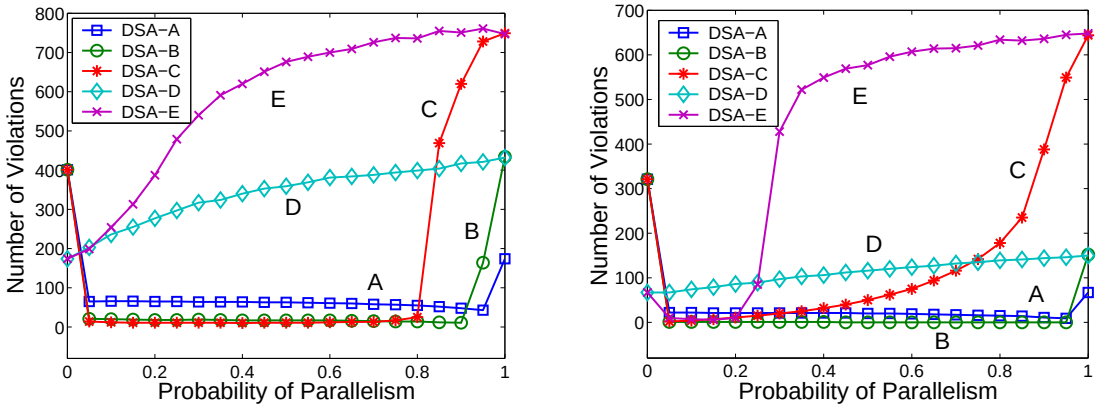


Figure 4: Solution quality phase transitions on graphs; $k = 8$ using 4 colors (left) and 5 colors (right).

that an agent will send a message to its neighbors after it changed its value (cf. Algorithm 1 and Table 1). In DSA-A, DSA-B and DSA-D, an agent may change its value if there is a conflict, and will not do so if it is currently at a state of a local minimum, while in DSA-C and DSA-E, an agent may probabilistically change its value at a local minimum state. Therefore, in general the communication cost at a node will go down if the agent moves to a better state, and go up otherwise. As a result, the overall communication cost will also follow similar trends. If the solution quality of DSA improves over time, so does its communication cost. Therefore, the communication cost is also controlled by the degree of parallel executions of the agents. The higher the parallel probability p is, the higher the communication cost will be.

We verified this prediction by experiments on grids and graphs, using the same problem instances as used for analyzing solution quality. Figure 5 shows the communication cost on grids with $k = 8$ using 4 colors (left) and 5 colors (right) after 1,000 steps. Comparing these these figures with those in Figure 2, it is obvious that solution quality and communication cost follow identical patterns. Furthermore, the communication cost on graphs (not shown here) follows similar patterns as those on grids.

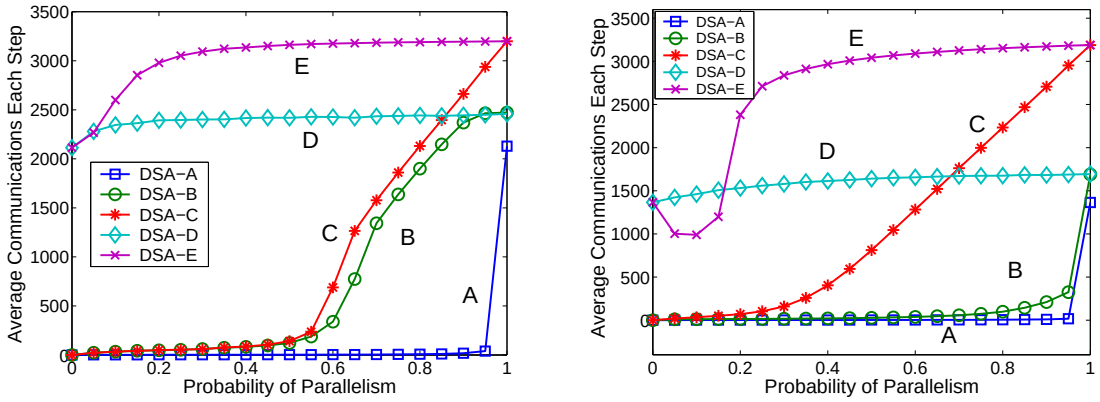


Figure 5: Communication phase transitions on grids with $k = 8$ using 4 colors (left) and 5 colors (right).

2.4 Breakout and Distributed Breakout

Distributed Breakout algorithm was based on a centralized predecessor. To better understand the distributed algorithm, we start with the centralized version. We then study the strength and weakness of DBA for graph coloring.

2.4.1 The algorithm

Algorithm 2 Sketch of DBA

```
set the local weights of constraints to one
value ← a random value from domain
while (no termination condition met) do
  exchange value with neighbors
  WR ← BestPossibleWeightReduction()
  send WR to neighbors and collect their WRs
  if (WR > 0) then
    if (it has the biggest improvement among neighbors) then
      value ← the value that gives WR
    end if
  else
    if (no neighbor can improve) then
      increase violated constraints' weights by one
    end if
  end if
end while
```

The breakout algorithm [79] is a local search method equipped with an innovative scheme of escaping local minima for CSP. Given a CSP, the algorithm first assigns a weight of one to all constraints. It then picks a value for every variable. If no constraint is violated, the algorithm terminates. Otherwise, it chooses a variable that can reduce the total weight of the unsatisfied constraints if its value is changed. If such a weight-reducing variable-value pair exists, the algorithm changes the value of a chosen variable. The algorithm continues the process of variable selection and value change until no weight-reducing variable

can be found. At that point, it reaches a local minimum if a constraint violation still exists. Instead of restarting from another random initial assignment, the algorithm tries to escape from the local minimum by increasing the weights of all violated constraints by one and proceeds as before. This weight change will force the algorithm to alter the values of some variables to satisfy the violated constraints.

Centralized breakout can be extended to distributed breakout algorithm (DBA) [103, 105]. Without loss of generality, we assign an agent to a variable, and assume that all agents have unique identifiers. Two agents are *neighbors* if they share a common constraint. An agent communicates only with its neighbors. At each step of DBA, an agent exchanges its current variable value with its neighbors, computes the possible weight reduction if it changes its current value, and decides if it should do so. To avoid simultaneous variable changes at neighboring agents, only the agent having the maximal weight reduction has the right to alter its current value. If ties occur, the agents break the ties based on their identifiers. The above process of DBA is sketched in Algorithm 2. For simplicity, we assume each step is synchronized among the agents. This assumption can be lifted by a synchronization mechanism [94].

In the description of [103, 105], each agent also maintains a variable, called *my-termination-counter* (MTC), to help detect a possible termination condition. At each step, an agent's MTC records the diameter of a subgraph centered around the agent within which all the agents' constraints are satisfied. For instances, an agent's MTC is zero if one of its neighbors has a violated constraint; it is equal to one when its immediate neighbors have no violation. Therefore, if the diameter of the constraint graph is known to each agent, when an agent's MTC is equal to the known diameter, DBA can terminate with the current agent values as a satisfying solution. However, MTCs may never become equal to the diameter even if a solution exists. There are cases in which the algorithm is not complete in that it cannot guarantee to find a solution even if one exists. Such a worst case depends on the structure of a problem, a topic of the next section. We do not include the MTC here to keep our description simple.

It is worth pointing out that the node, or agent, identifiers are not essential to the algorithm. They are only used to set up a priority between two competing agents for tie breaking. As long as such priorities exists, node identifiers are not needed.

2.4.2 Completeness and Complexity

Distributed algorithms are generally incomplete in that they cannot guarantee to find a solution even if one exists. It is also generally difficult to show the completeness of a distributed algorithm and to analyze its complexity. In this section, we consider the completeness and complexity of DBA on a special class of problems, i.e., distributed graph coloring. Here, the complexity is defined as DBA's number of synchronized distributed steps. In one step, value changes at different nodes are allowed while one variable can change its value at most once. We also use variables, nodes and agents interchangeably in our discussion.

Acyclic graphs: First notice that acyclic graphs are 2-colorable. Thus, coloring acyclic graphs is less constrained with more colors. Therefore, it is sufficient to consider acyclic graphs with two colors. To simplify our discussion, we first consider chains, which are special acyclic graphs. The results on chains will also serve as a basis for trees.

Chains We will refer to the combination of variable values and constraint weights as a *problem state*, or *state* for short. A *solution* of a constraint problem is a state with no violated constraint. We say two states are *adjacent* if DBA can move from one state to the other within one step.

Lemma 2.1 *DBA will not visit the same problem state more than once when 2-coloring a chain.*

Proof: Assume the opposite, i.e., DBA can visit a state twice in a process as follows, $S_x \rightarrow S_y \rightarrow \cdots \rightarrow S_z \rightarrow S_x$. Obviously no constraint weight is allowed to increase at any state on this cycle. Suppose that node x changes its value at state S_x to resolve a conflict C involving x. In the worst case a new conflict at the other side of the node will be created. C is thus "pushed" to the neighbor of x, say y. Two possibilities exist. First, C is resolved at y or another node along the chain, so that no state cycle will form. Second, C returns to x, causing x to change its value back to its previous value. Since nodes are ordered, i.e., they have prioritized identifiers, violations may only move in one direction and C cannot return to x from y without changing a constraint weight. This means that C must move back to x from another path, which contradicts the fact that the structure is a chain. □

Lemma 2.2 *DBA can increase a constraint weight to at most $\lfloor n/2 \rfloor$ when coloring a chain of n variables using at least two colors.*

Proof: The weight of the first constraint on the left of the chain will never change and thus remain at one, since the left end node can always change its value to satisfy its only constraint. The weight of the second constraint on the left can increase to two at the most. When the weight of the second constraint is two and the second constraint on the left is violated, the second node will always change its value to satisfy the second constraint because it has a higher weight than the first constraint. This will push the violation to the left end node and force it to change its value and thus resolve the conflict. This argument can be inductively applied to the other internal nodes and constraints along the chain. In fact, it can be applied to both ends of the chain. So the maximal constraint weight on the chain will be $\lfloor n/2 \rfloor$. □

Immediate corollaries of this lemma are the best and worst arrangements of variable identifiers. In the best case, the end nodes of the chain should be most active, always trying to satisfy the only constraint, and resolving any conflict. Therefore, the end nodes should have the highest priority, followed by their neighbors, and so on to the middle of the chain. The worst case is simply the opposite of the best case; the end nodes are most inactive and have the lowest priority, followed by their neighbors, and so on.

Theorem 2.1 *DBA terminates in at most n^2 steps with a solution, if it exists, or with an answer of no solution, if it does not exist, when coloring a chain of n variables using at least two colors.*

Proof: As a chain is always 2-colorable, the combination of the above lemmas gives the result for a chain with nodes of domain sizes at least two. It is possible, however, that no solution exists if some variables have fixed values or colors. In this case, it is easy to create a conflict between two nodes with domain size one, which will never be resolved. As a result, the weights of the constraints between these two nodes will be raised to n. If each agent knows the chain length n, DBA can be terminated when a constraint

weight is more than n. (In fact, the chain length can be computed in $O(n)$ steps as follows. An end node first sends number 1 to its only neighbor. The neighboring node adds one to the number received and then passes the new number to the other neighbor. The number reached at the other end of the chain is the chain length, which can be subsequently disseminated to the rest of the chain. The whole process takes $2n$ steps.) Furthermore, a node needs at most $n-1$ steps to increase a constraint weight. This worst case occurs when a chain contains two variables at two ends of the chain which have the lowest priorities and unity domain sizes so neither of them can change its value. On such a chain, a conflict can be pushed around between the two end nodes many time. Every time a conflict reaches an end node, the node increases the constraint weight to push the conflict back. Since a constraint weight will be no more than n, the result follows. □

A significant implication of these results is a termination condition for DBA for coloring a chain. If DBA does not find a solution in n^2 steps, it can terminate with the conclusion that no solution exists. This new termination condition and DBA's original termination condition of my-termination-counter guarantee DBA to terminate on a chain.

Trees The key to the proof for coloring chain and tree structures is that no cycle exists in an acyclic graph, so that the same conflict cannot return to a node without increasing a constraint weight.

The arguments on the maximal constraint weight for coloring chains hold for general acyclic graphs or trees. First consider the case that each variable has a domain size at least two. In an acyclic graph, an arbitrary constraint (link) C connects two disjoint acyclic graphs, G_1 and G_2. Assume G_1 and G_2 have n_1 and n_2 nodes, respectively, and $n_1 \leq n_2$. Then the maximal possible weight W on C cannot be more than n_1, which is proven inductively as follows. If the node v associated with C is the only node of G_1, then the claim is true since v can always accommodate C. If G_1 is a chain, then the arguments for Lemma 2.2 apply directly and the maximal possible weight of a constraint is the number of links the constraint is away from the end variable of G_1. If v is the only node in G_1 that connects to more than one constraint in G_1, which we call a branching node, then a conflict at C may be pushed into G_1 when the weight of C is greater than the sum of the weights of all constraints in G_1 linked to v, which is at most equal to the number of nodes of G_1. The same arguments equally apply when v is not the only branching node of G_1. Therefore, the maximal constraint weight is bounded by n.

The worst-case complexity can be derived similarly. A worst case occurs when all end variables of an acyclic graph have fixed values, so that a conflict may never be pushed out of the graph. A constraint weight can be bumped up by one after a conflict has traveled from an end node to other end nodes and back, within at most n steps.

Based on these arguments, we have the following result.

Theorem 2.2 *DBA terminates in at most n^2 steps with either an optimal solution, if it exists, or an answer of no solution, if it does not exist, when coloring an acyclic graph with n nodes with at least two colors.*

The above completeness result can be directly translated to centralized breakout algorithm, leading to its completeness of coloring acyclic graphs as well. Moreover, since each step in DBA is equivalent to n steps in the centralized algorithm, each of which examines a distinct variable, the complexity result on

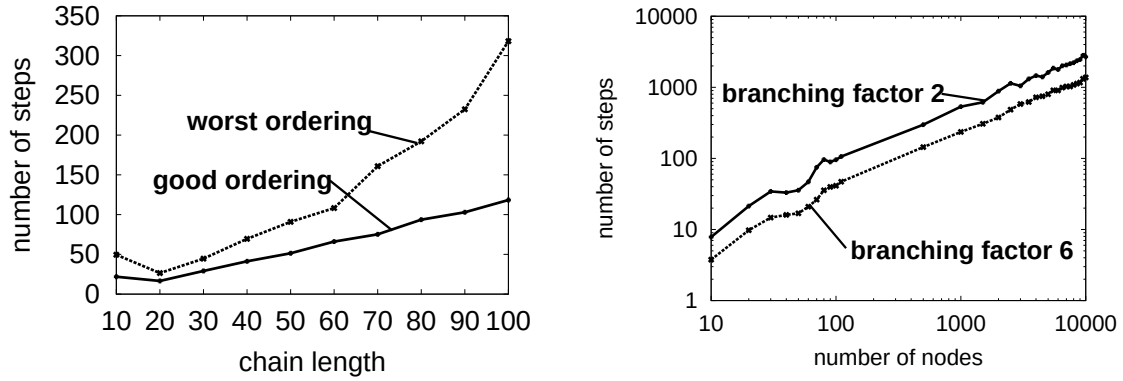


Figure 6: The number of steps taken by DBA on chains with the best and worst variable identifier arrangements (left) and on trees with worst identifier arrangements (right).

DBA also means that the worst-case complexity of the centralized algorithm is $O(n^3)$. These analytical results reveal the superiority of centralized breakout algorithm and DBA over conventional local search methods of coloring acyclic graphs, including the distributed stochastic algorithm discussed in Section 2.3, which are not complete even on a chain.

Our experimental results also show that the number of steps taken by DBA is much smaller than the n^2 upper bound, as shown in Figure 6. In our experiments, we used different size chains and trees and averaged the results over 100 random trials. We considered the best- and worst-case identifier arrangements for chains (Figure 6 left) and worst-case arrangement for trees (where more active nodes are closer to the centers of the trees) with different branching factors. As the figure shows, the average number of steps taken by DBA is near linear for the worst-case identifier arrangement, and the number of steps is linear on trees with a worst-case identifier arrangement (Figure 6 right). Furthermore, for a fixed number of nodes the number of steps decreases inversely when branching factors of the trees increase. In short, DBA is efficient on coloring acyclic graphs.

Cyclic graphs: Unfortunately, DBA is not complete on cyclic constraint graphs. This will include non-binary problems as they can be converted to binary problems with cycles. This is also the reason that breakout algorithm is not complete on Boolean satisfiability with three variables per clause [79], which is equivalent to a constraint with three variables.

When there are cycles in a graph, conflicts may walk on these cycles forever. To see this, consider a problem of coloring a ring with an even number of nodes using two colors (black and white), as shown in Figure 7, where the node identifiers and constraint weights are respectively next to nodes and edges. Figure 7(1) shows a case where two conflicts appear at locations between nodes 1 and 3 and between nodes 4 and 5, that are not adjacent to each other. The weights of the corresponding edges are increased accordingly in Figure 7(2). As node 1 (node 4) has a higher priority than node 3 (node 5), it changes its value and pushes the conflict one step counter-clockwise in Figure 7(3). The rest of Figure 7 depicts the subsequent steps until all constraint weights have been increased to 2. This process can continue forever with the two conflicts moving in the same direction on the chain at the same speed, chasing each other endlessly and making DBA incomplete.

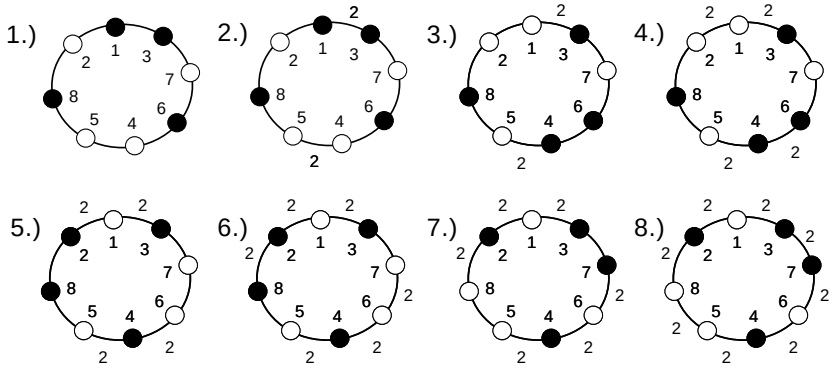


Figure 7: A worst case of DBA for coloring a ring.

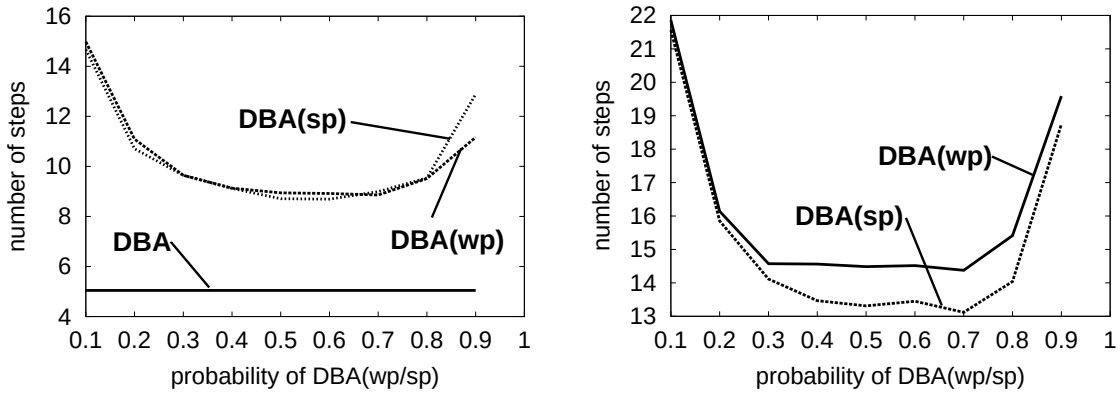


Figure 8: Steps taken by DBA and variants on the example of Figure 7 with random initial assignments (left) and the specific assignment of Figure 7 (right).

2.4.3 Stochastic Variations

A lesson that can be learned from the above worst-case scenario is that conflicts should not move at the same speed. We thus introduce randomness to alter the speeds of possible conflict movements on cycles of a graph. This stochastic feature may increase DBA's chances of finding a solution possibly with a penalty on convergence to solution for some cases.

DBA(wp) and DBA(sp): We can add randomness to DBA in two ways. In the first, we use a probability for tie breaking. The algorithm will proceed as before, except that when two neighboring variables have the same improvement for the next step, they will change their values probabilistically. This means that both variables may change or not change, or just one of them. We call this variation weak probabilistic DBA, denoted as DBA(wp).

In the second method, which was inspired by the distributed stochastic algorithm [25, 27, 116], a variable will change if it has the best improvement among its neighbors. However, when it can improve but the improvement is not the best among its neighbors, it will change based on a probability. This variation is more active than DBA and the weak probabilistic variation. We thus call it strong probabilistic DBA, DBA(sp) for short.

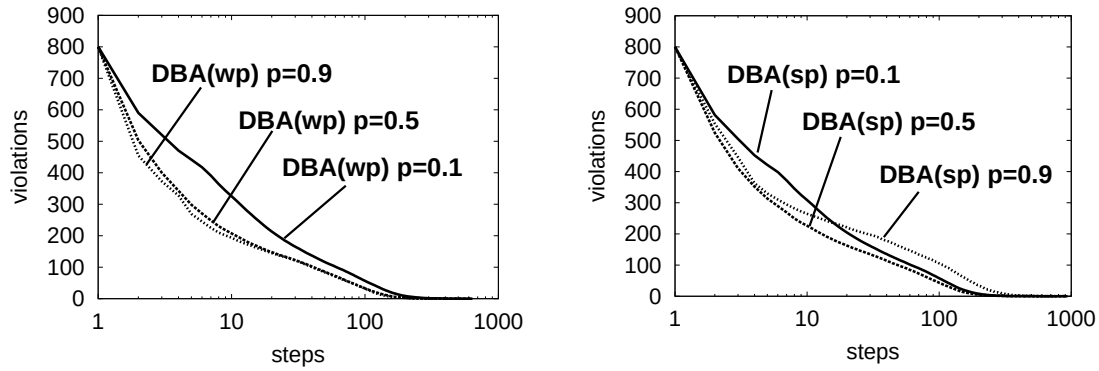


Figure 9: DBA(wp) and DBA(sp) on grid 20×20 and $k = 4$.

One favorable feature of these variants is that no variable identifiers are needed, which may be important for some applications where node identifiers across the whole network is expensive to compute. Moreover, these variants give two families of variations to DBA, depending on the probabilities used. It will be interesting to see how they vary under different parameters, the topic that we consider next.

DBA(wp) versus DBA(sp): We first study the two variants on the example of coloring an 8-node ring of Figure 7. In the first set of tests, node identifiers and initial colors are randomly generated and 10,000 trials are tested. DBA is unable to terminate on 15% of the total trials after more than 100,000 steps[3], while on the other 85% of the trials DBA finds a solution after 5 steps on average as shown in Figure 8(left). In contrast, DBA(wp) and DBA(sp) always find solutions but require almost twice as many steps on average with the best probability around 0.6.

In the second set of tests, we use the exact worst-case initial assignment as shown in Figure 7. As expected, DBA failed to terminate. DBA(wp) and DBA(sp) find all solutions on 1,000 trials. Since they are stochastic, each trial may run a different number of steps. The average number of steps under different probability is shown in Figure 8(right).

Next we study these two families of variants on grids, graphs and trees. We consider coloring these structures using 2 colors. For grids, we consider 20×20, 40×40, and 60×60 grids with connectivities equal to $k = 4$ and $k = 8$. To simulate infinitely large grids in our experiments, we remove the grid boundaries by connecting the nodes on the top to those on the bottom as well as the nodes on the left to those on the right of the grids to create $k = 4$ grid. For $k = 8$ grid, we further link a node to four more neighbors, one each to the top left, top right, bottom left and bottom right. This renders the problem overconstrained for two-coloring. Hence, the algorithms may only try to improve the solution quality by minimizing the number of violated constraints.

The results of 20×20 grids with $k = 4$ are shown in Figure 9, averaged over 2,000 trials. As the figures show, the higher the probability the better DBA(wp)'s performance. For DBA(sp) $p = 0.5$ is the best probability.

We generate 2,000 graphs with 400 nodes with an average connectivity per node equal to $k = 4$ and

[3]Our additional tests also show that DBA's failure rate decreases as the ring size increases.

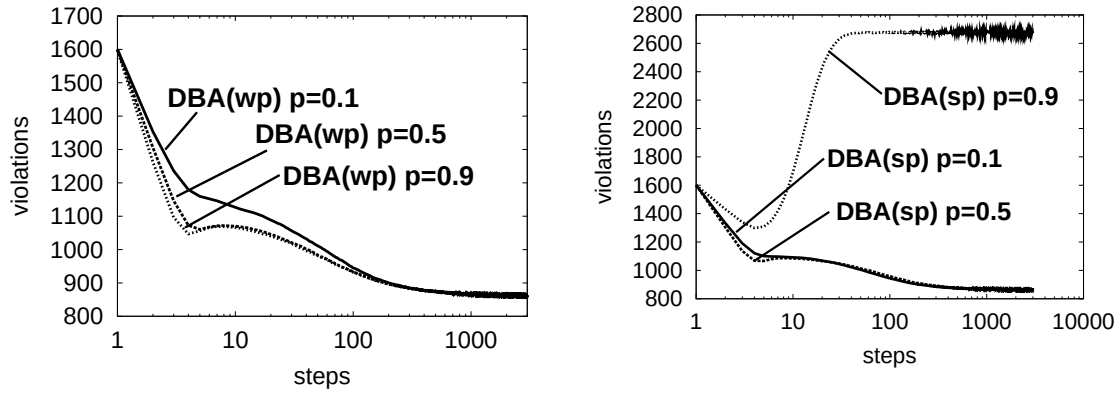


Figure 10: DBA(wp) (left) and DBA(sp) (right) on graph with 400 nodes and k=8.

$k = 8$ by adding, respectively, 1,600 and 3,200 edges to randomly picked pairs of unconnected nodes. These two graphs are generated to make a correspondence to the grid structures of $k = 4$ and $k = 8$ considered previously, except that both random graphs are not two-colorable. All algorithms are applied to the same set of graphs for a meaningful comparison. Figure 10 shows the results on graphs with $k = 8$. There is no significant difference within the DBA(wp) family. However, DBA(sp) with large probabilities can significantly degrade to very poor performance, exhibiting a phenomenon similar to phase transitions. Since DBA(sp) with high probability is close to distributed stochastic algorithm [25, 27, 116], the results here are in line with those of [116].

We also consider DBA(wp) and DBA(sp) on random trees with various depths and branching factors. Due to space limitations, we do not include detailed experimental results here, but give a brief summary. As expected, they all find optimal solutions for all 10,000 2-coloring instances. Within DBA(wp) family, there is no significant difference. However, DBA(sp) with a high probability has a poor anytime performance.

Combining all the results on the constraint structures we considered, DBA(sp) appears to be a poor algorithm in some cases, especially when its probability is very high.

DBA(wp) and DBA(sp) versus DBA: The remaining issue is how DBA(wp) and DBA(sp) compare with DBA. Here we use the best parameters for these two variants from the previous tests and compare them directly with DBA. We average the results over the same sets of problem instances we used in Section 2.4.3. Figures 11, 12 and 13 show the experimental results on grids, random graphs and trees, respectively. With their best parameters, DBA(wp) and DBA(sp) appear to be compatible with DBA. Furthermore, as discussed earlier, DBA(wp) and DBA(sp) increase the probability of convergence to optimal solutions. DBA(wp), in particular, is a better alternative in many cases if its probability is chosen carefully. Stochastic features do not seem to impair DBA's anytime performance on many problem structures and help overcome the problem of incompleteness of DBA on graphs with cycles.

2.5 Comparative Analysis and Application

We now directly compare DBA and DSA on multi-coloring problems generated from the scan scheduling problem discussed in Section 2.2. In our experiments, we set the sensing radius of a sensor to one unit, and

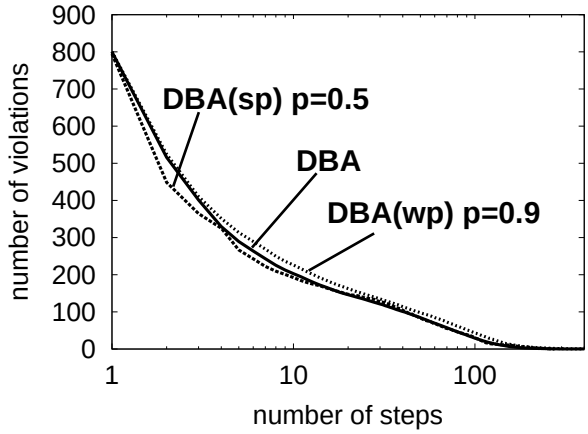


Figure 11: DBA and random DBAs on grid 20×20 and $k = 4$.

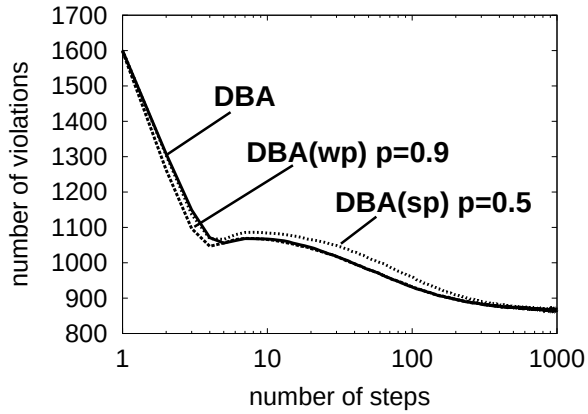


Figure 12: DBA and random DBAs on graph with 400 nodes and k=8.

used a square of 10×10 units as the area to be monitored. The number of sensing sectors is set to three to match our hardware system. We randomly and uniformly placed a fixed number of sensors with arbitrary orientations in the square. We then converted these problems into multi-coloring problems as described in Section 2.2. We experimented with different values of maximum allowed colors T, which correspond to the cycle lengths of scanning, and different sensor activation ratios α, which determin how often the sensors will be active within a scanning cycle. In the following, we report the results using $T = 6$ and $T = 18$ with $\alpha = 2/3$.

For DSA, we changed its probability p of parallel executions from 0.1 to 0.99, with an increment of 0.01. We used 100 instances for each p. We evaluated the performance of DSA and DBA when they have reached relatively stable states. Specifically, we ran DSA and DBA to the point where their performance does not change significantly from one step to the next. On all network sizes we considered, these algorithms' performance seems to be stabilized after 256 steps. Similar results have been observed after 1024, 2048 and longer steps. In the rest of this section, we report the results at 256 steps.

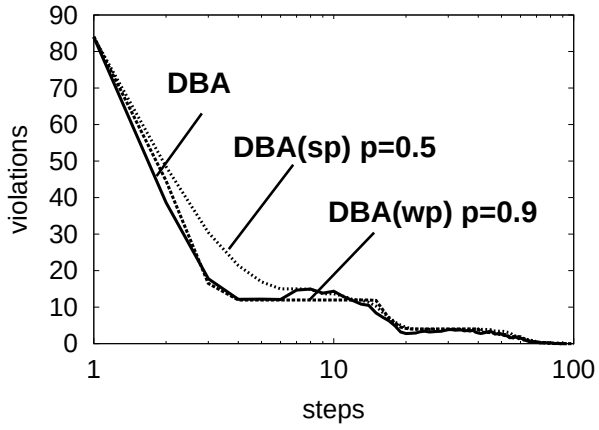


Figure 13: DBA and random DBAs on tree with depth $d = 4$ and branching factor $k = 4$.

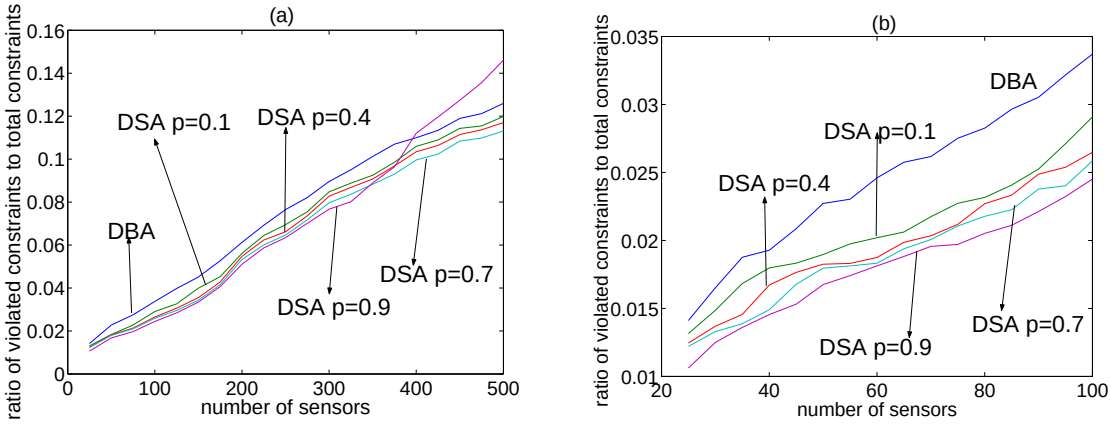


Figure 14: DSA vs. DBA in terms of number of sensors, $T = 6$.

2.5.1 Solution quality in terms of network sizes

Since one of our ultimate objectives is to choose DBA or DSA to solve our distributed scan scheduling problem, we need to investigate the relationship between the quality of the schedules found by these two algorithms and the properties of the underlying networks. To this end, we experimentally compared DBA and DSA on multi-coloring problems produced from sensor networks of various sizes. We changed the density of multi-coloring graphs by changing the number of sensors N. The solution quality is the total weight of violated soft constraints normalized by the total weight of soft constraints, measured at 256 steps of the algorithms' executions when their performances are relatively stable.

We run DBA and DSA with four different representative probabilities p of parallel executions, 0.1, 0.4, 0.7 and 0.9. We varied the density or number of sensors and compared the quality of the colorings that DSA and DBA produced. We changed the number of sensors from 25 to 100 with an increment of 5 sensors, and from 100 to 500 with an increment of 25 sensors. We averaged the results over 100 random problem instances for each fixed number of sensors. Figure 14 shows the result on $T = 6$. The horizontal axes in the figures are the numbers of sensors, and the vertical axes are the normalized solution

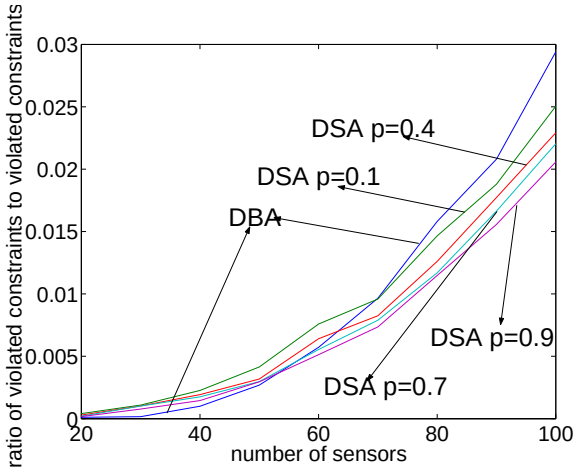


Figure 15: DSA vs. DBA in terms of number of sensors, $T = 18$.

quality after 256 steps. Longer executions, such as 512 and 1024 steps, exhibit almost identical results. Figure 14(a) shows the result in the whole range of 25 to 500 sensors and Figure 14(b) expands the results of Figure 14(a) in the range of 25 to 100 sensors,

As analyzed in Section 2.4, DBA may perform better than DSA on underconstrained problems, especially acyclic graphs. An underconstrained scan scheduling problem may be created when more colors are available. Indeed, when we increase the number of allowed colors (targeting schedule cycle length) to eighteen ($T = 18$), DBA outperforms DSA on sparse networks with less than 50 sensors. This result is shown in Figure 15, where each data point is averaged over 100 trials.

Based on the experimental results, we can reach three conclusions. First, DBA typically performs worse than DSA when its degree of parallelism is not too high in the range of 25 to 500 sensors. Second, when the sensor density increases, the performance of DSA may degenerate, especially if its degree of parallelism p is high. For instance, DSA with $p = 0.9$ becomes the worst of all when there are more than 400 sensors (Figure 14). This means that the denser the sensor networks are, the smaller the parallel degree p should be. The degenerated performance of DSA with a large p is mainly due to its phase-transition behavior revealed in the previous section. When the sensor density increases, more constraints will be introduced into the inherited constraints of the scan scheduling problem, so that the problem becomes overconstrained. As indicated in the phase-transition section, DSA's phase-transition behavior appears sooner when overall constraints are tighter. Third, in the underconstrained region, a higher degree of parallelism is preferred to a lower degree.

2.5.2 Anytime performance

An important feature of our targeting sensor network for object detection is real-time response. High real-time performance is important, especially for systems with limited computation and communication resources in which it may be disastrous to wait for the systems to reach stable or equilibrium states. This is particularly true for our sensor-based system for object detection and mobile object tracking. Therefore,

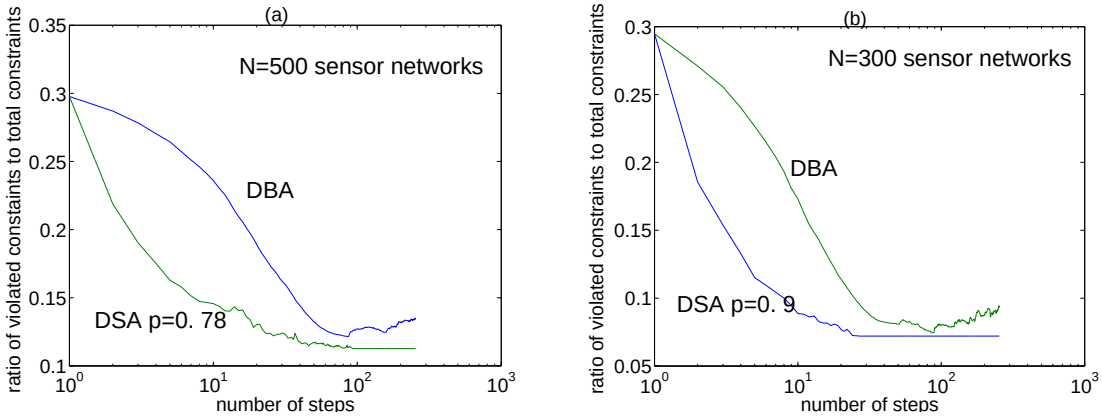


Figure 16: Anytime performance of DSA and DBA in dense sensor networks, $T = 6$.

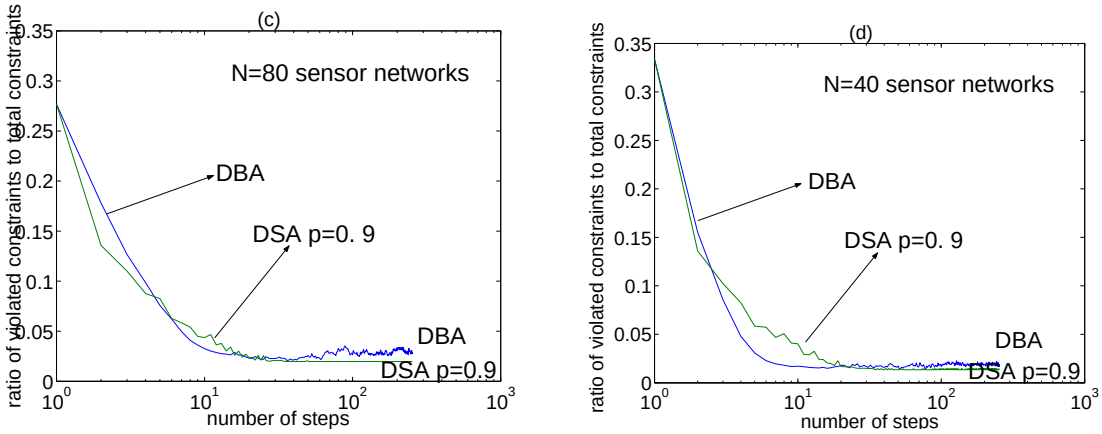


Figure 17: Anytime performance of DSA and DBA in sparse sensor networks, $T = 6$.

the algorithm for the distributed scan scheduling must have anytime property, i.e., the algorithm can be stopped at anytime during its execution and is able to provide a feasible solution at that point. Fortunately, DBA and DSA can both be used for this purpose because the hard constraints internal to individual sensors (a sensor cannot scan its two sectors at the same time) are always maintained.

In the rest of this section, we directly compare DBA and DSA as anytime algorithms. We use the same set of experimental conditions and parameters as in the previous sections, i.e., sensors have three sectors and are randomly and uniformly placed on a 10×10 grid, with results averaged over 100 trials.

We first consider dense networks with $N = 500$ and $N = 300$ sensors. Based on the phase-transition results in Section 2.3.2, DSA performs the best with $p = 0.9$ and $p = 0.78$ for the networks of $N = 500$ and $N = 300$ nodes, respectively. We used these parameters in our experiments. The experimental results are in Figure 16. As the results show, DSA performs much better than DBA in both anytime performance and final solution quality.

We now consider sparse sensor networks, using networks with $N = 80$ and $N = 40$ sensors as representatives. As discussed earlier, a higher degree of parallelism should be used on sparse graphs, we

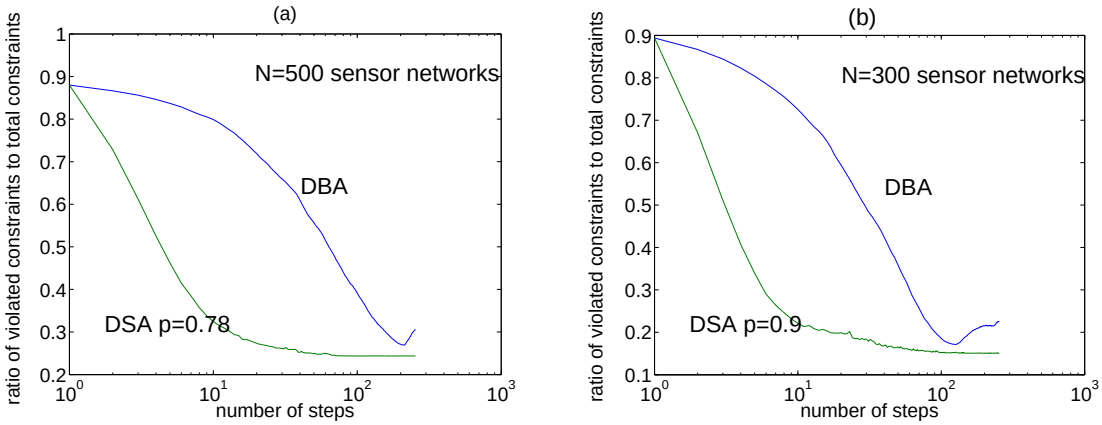


Figure 18: Anytime performance of DSA and DBA in dense sensor networks, $T = 18$.

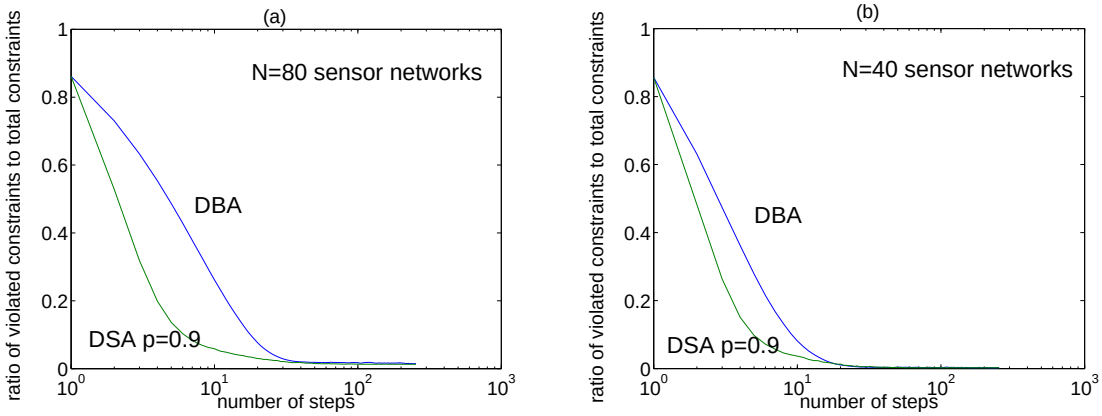


Figure 19: Anytime performance of DSA and DBA in sparse sensor networks, $T = 18$.

thus use $p = 0.9$ for our two sparse networks. The results, averaged over 100 trials, are in Figure 17. Clearly, DBA and DSA exhibits similar performance, with DSA being able to produce slightly better solutions at the end.

To complete our analysis, we also compared DSA and DBA on the same sets of instances, but with 18 available colors ($T = 18$). The results on dense and sparse sensor networks are included in Figures 18 and 19, respectively. Interestingly, DBA's anytime performance degenerates, compared to that using $T = 6$.

In summary, as far as solution quality (anytime and final solutions) is concerned, our experimental results indicate that DSA should be adopted for the distributed scan scheduling problem.

2.5.3 Communication Cost

We have so far concentrated on the solution quality of DBA and DSA without paying any attention to their communication cost. As mentioned earlier, communication in a sensor network has an inherited delay and could be unreliable in most situations. Therefore, a good distributed algorithm should require a small number of message exchanges.

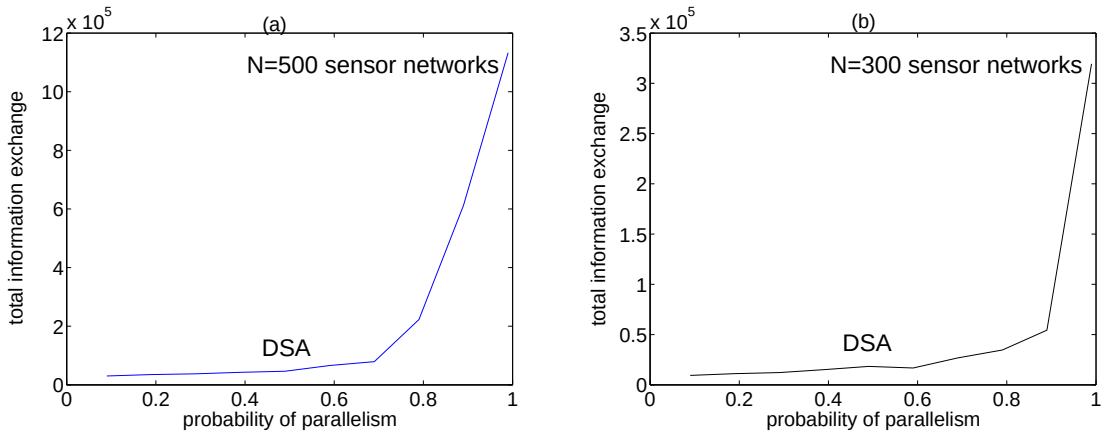


Figure 20: Communication-cost phase transitions of DSA on scan scheduling, $T = 6$.

In each step of DBA, an agent announces its best possible conflict reduction to its neighbors and receives from the neighbors their possible weight reductions. Thus, the number of messages sent and received by an agent in each step of DBA is no less than the number of its neighbors, and the total number of messages exchanged in each step is more than a constant for a given network.

In contrast, an agent may not have to send a message in a step in DSA if it does not change its value. In an extreme case, an agent will not change its value if it is at an local minima. If solution quality of DSA improves over time, its communication cost will reduce as well. In principle, the communication cost of DSA is correlated to its solution quality. The better the current solution, the less the number of messages. In addition, the communication cost is also related to the degree of parallel executions of the agents. The higher the parallel probability p is, the higher the communication cost will be. As shown in Section 2.3, the communication cost of DSA goes hand-in-hand with its solution quality and also experiences a similar phase-transition or threshold behavior on regular coloring problems. Figure 20 shows the phase-transition behavior of DSA's communication cost on the $N = 500$ and $N = 300$ sensor networks using $T = 6$ that we studied before. Here we considered the accumulative communication cost of all 256 steps. This result indicates that the degree of parallelism must be controlled properly in order to make DSA effective. Similar phase-transition patterns have been observed when we use $T = 18$.

We now compare DSA and DBA in terms of communication cost. Figure 21 shows the results evaluating DBA and DSA with probability $p = 0.78$ on $N = 500$ networks using $T = 6$ and $T = 18$, averaged over 100 trials. The figures plot the average numbers of messages exchanged in DSA and DBA at a particular step. Clearly, DSA has a significant advantage over DBA on communication cost. The large difference on communication cost between DSA and DBA will have a significant implication on how these two algorithms can be used in real sensor networks, especially when the sensors are connected through delayed, unreliable and noisy wireless communication. For our particular application and system where communication was carried out by radio frequencies, DBA's high communication cost makes it noncompetitive.

In summary, in terms of solution quality and communication cost, DSA is preferable over DBA for our

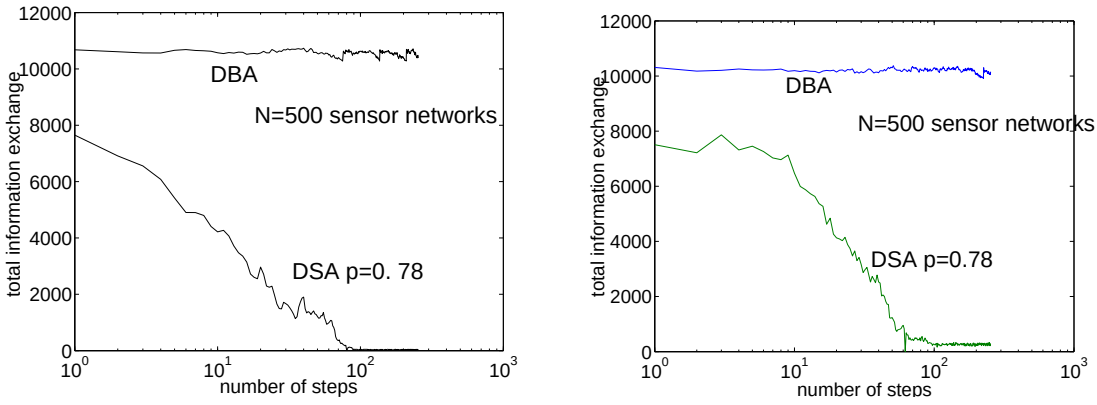


Figure 21: Communication cost of DSA and DBA, $T = 6$ (left) and $T = 18$ (right).

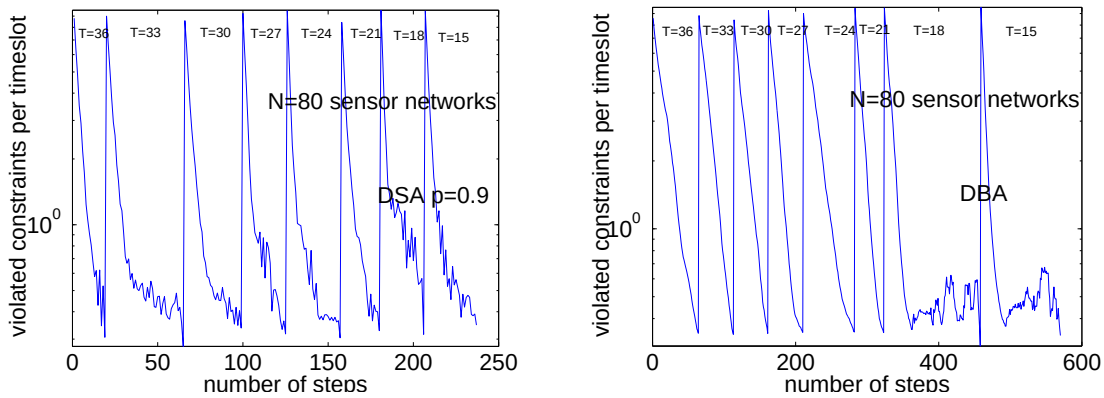


Figure 22: Finding best possible schedule using DSA (top) and DBA (bottom), $N = 80$.

distributed scan scheduling if DBA's degree of parallelism is properly controlled.

2.5.4 Solving Scheduling Problem

Based on the results from Sections 2.3.2 to 2.5.3, we now apply DSA and DBA to dealing with two related problems at the same time, finding the shortest scan cycle length T and obtaining a good schedule given the shortest cycle length T.

To this end, we run DSA and DBA in iterations, starting with an initially large T. T is reduced after each iteration. Given a T in an iteration, DSA or DBA searches for a schedule of a quality better than a predefined threshold Q. The iteration stops whenever such a schedule is found within a fixed number of steps, and a new iteration may start with a smaller T.

In our simulation, we checked the quality of the current schedule after each simulated step. As soon as the quality of the current schedule exceeds the given threshold Q, we terminate the current iteration. This is equivalent to having an agent compute the global state of a distributed system, a method infeasible for our completely distributed system. We use this mechanism here simply to evaluate the performance of DSA and DBA.

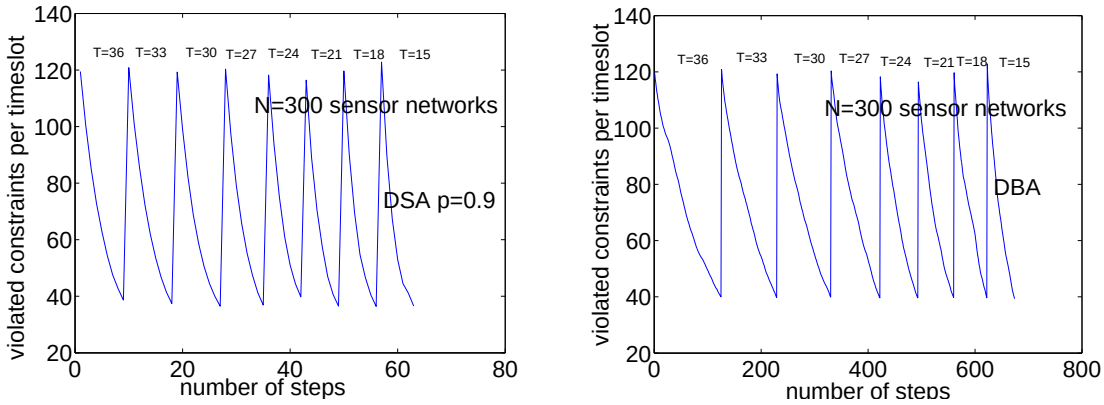


Figure 23: Finding best possible schedule using DSA (top) and DBA (bottom), $N = 300$.

Figures 22 and 23 show the results on two networks, one with $N = 80$ sensors and the other with $N = 300$. In our experiments, we fixed the sensor activation ratio at $\alpha = 2/3$, used initial $T = 36$, and reduced T by three after each iteration, which ran a maximum of 256 steps. The threshold for schedule quality was set to $Q = 0.01$ for $N = 80$ and $Q = 40$ for $N = 300$. As the results show, DSA is superior to DBA. On the $N = 80$ network (Figure 22), DSA finds a targeting schedule of length $T = 15$ in 242 steps, while DBA needs 588 steps. On the $N = 300$ network (Figure 23), DSA takes 64 steps, while DBA uses 691 steps, which is an order of magnitude difference.

In summary, our results clearly show that DSA is superior to DBA on the distributed scan scheduling problem. If communication cost is also a concern, DSA is definitely the algorithm of choice for the problem.

2.6 Related Work and Discussions

The basic idea of distributed stochastic search must have been around for some time. A similar idea was used by Pearl in distributed belief update [83]. The idea was directly used for distributed graph coloring in [25, 27]. DSA-B considered here is the same as CFP in [27]. However, [25, 27] failed to reveal phase transitions discussed in this section. The idea was also studied using spin glasses models [67] where phase transitions were characterized. Phase transitions in distributed constraint problem solving was also reported in [42].

This research extends the existing work of DSA in many different ways. It proposes two variations to the basic DSA. It systematically studies observation-based, distributed stochastic search for distributed coordination and provides an experimental, qualitative analysis on the relationship among the degree of parallelism, problem constrainedness, solution quality and overall system behavior such as phase transitions. It also demonstrates that phase transitions exist in many different problems and problem structures and they persist when the degree of parallelism changes. Notice that the phase transitions considered in this section are different from phase transitions of graph coloring problems [14]. Here we studied the phase-transition behavior of distributed search algorithms, which needs not be phase transitions of the

coloring problems we considered.

Other related algorithms include complete algorithms for DisCSP, such as the asynchronous weak-commitment (AWC) search algorithm [103, 104], and for DisCOP. These complete algorithms are important for distributed constraint solving. Comparing to DBA and DSA, however, they require longer running time and usually have worse anytime performance, making them inferior for real-time applications where optimal solutions may be too costly or may not be necessary. In addition, these complete algorithms need a sufficiently large amount of memory to record the states (agent views) that an agent has visited in order to avoid revisiting a state multiple times so as to make the algorithms converge to a solution if it exist. In contrast, DSA and DBA are able to reach near optimal solutions quickly without additional memory. This feature, along with their good anytime performance, made DSA and DBA attractive applications in sensor networks where memory is a critically limited resource. In addition, on coloring acyclic graphs, DBA is complete and has a low polynomial complexity, making it an alternative algorithm for optimal solution in such a case.

DSA differs from DBA, AWC and Adopt by the notion of uniformness. A distributed algorithm is called uniform if all nodes execute the same procedure and two nodes do not differ from each other [94]. Therefore, DSA is a uniform algorithm since the nodes do not have identifications and they all execute the same set of instructions. However, DBA, AWC and Adopt are not uniform because the nodes in these algorithms need to have identifications to differ from one another and to set priorities to decide what to execute next.

We need to emphasize that the notion of uniformness for distributed algorithms has a practical importance for applications in sensor networks. In a typical application in sensor networks, sensors may have to dynamically organize to form a system. The use of identifications of and priorities among nodes (sensors) will introduce prohibitive barriers on what systems a set of sensors and a given placement can form. The fact that DSA is a uniform algorithm further supports the conclusion from the comparison results in this section that DSA is the algorithm to choose over DBA.

2.7 Conclusions

We were motivated in this research to apply the framework of multiagent systems and the techniques of distributed constraint problem solving to resource bounded, anytime, distributed constraint problems in sensor networks. Our specific applications include the detection of mobile objects and the detection of material damage in real time using distributed sensors and actuators. We first formulated these problems as distributed multiple coloring problems with the objective of minimizing the number of violated constraints.

To cope with limited resources and to meet the restricted requirement of anytime performance, we were interested in those distributed algorithms that have low-overhead on memory and computation for solving distributed constraint optimization problems. We focused particularly on the distributed stochastic algorithm (DSA) [25, 27, 67, 83] and the distributed breakout algorithm (DBA) [79, 103, 105], two existing distributed algorithms that fit into the category of low-overhead distributed algorithms. We analyzed and compared DSA and DBA on distributed multiple coloring problems that were generated from our

distributed scheduling problems in sensor networks. We specifically investigated the relationship among the degree of parallel executions, problem constrainedness, and DSA's behavior and performance. We showed that DSA exhibits a threshold behavior similar to phase transitions in which its performance, in terms of both solution quality and communication cost, degrades abruptly and dramatically when the degree of agents' parallel execution increases beyond a critical point. We also studied the completeness and complexity of DBA on distributed graph coloring problems, showing that DBA is complete and has low polynomial complexity on coloring acyclic graphs. However, DBA is not complete in general. We also introduced randomization schemes to DBA to improve its worst case performance. Finally, we directly compared DSA and DBA on our application problems of distributed scheduling problems in sensor networks. We showed that if controlled properly, DSA is significantly superior to DBA, finding better solutions with less computational cost and communication overhead. For distributed scheduling problems such as the ones considered in this section, DSA is the algorithm of choice.

One important lession we have learned from this research was that full syncronization may not be absolutely necessary in achieving high degree of optimality in distributed environments. This is even more so when computational and communicational resources are critically limited and constrained, which appear very often in Autonamous Negotiating Teams domains.

3 Analysis of Negotiation Protocols by Distributed Search

Negotiation is an important mechanism for coordination and collaboration in multi-agent systems. It is particularly effective for systems consisting of self-interested agents, each of which may have different objectives to achieve and different restrictions to abide to. It is perhaps also prevalent in applications where privacy of individual agents needs to be protected. Due to its importance, negotiation has been studied for quite some time, and many different negotiation strategies and protocols have been proposed and developed [11, 2, 17, 23, 24, 62]. Cooperative negotiation has been studied to solve difficult distributed problems such as distributed conflict resolution [2, 62], distributed task allocation [11, 24], and distributed resource allocation [17]. In all these domains, multiple agents share some common resources, e.g., communication channels and CPU times, and can mutually benefit from one another by cooperatively scheduling the resources. For these problems, agents only have the information about their local tasks and resources. They gradually become aware of the global information through negotiation and solve the global problem by individually solving sub-problems and integrating the solutions to sub-problems into a globally consistent solution [64]. For some applications, agents may also need to assess and refine the global solutions into global optimal solutions to make the best use of common resources.

However, global optimal solutions are very hard to achieve through cooperative negotiation in a distributed setting, due to its computational and communicational complexity. Furthermore, most existing negotiation protocols are complex and their features are difficult to characterize. To our knowledge, we have not seen a published work that analyzes a negotiation protocol in depth to understand important issues of a negotiation method, such as its completeness, complexity and scalability.

In this section, we propose an experimental approach to analyzing negotiation methods. Our strategy consists of two steps. In the first step, we formulate the distributed problems solved by a negotiation method by distributed constraint satisfaction/optimization problems, and capture the negotiation protocol as a distributed constraint search. In the second step, we study the properties of the negotiation protocol by analyzing the derived search algorithm. Note that to a larger extent, the idea of viewing negotiation as distributed search is not completely new. Indeed, it has been suggested that distributed AI can be viewed as distributed search [64]. Nevertheless, we not only view negotiation as distributed search in this research, but take this view one step further and directly apply a search algorithm to capture the essential features of a negotiation protocol and analyze its properties and performance.

In this research, we specifically focus on a resource allocation problem underlying a mobile object tracking problem using distributed sensors and a recently developed negotiation protocol, Scalable Protocol for Anytime Multi-level negotiation (SPAM) [68], for this problem. The SPAM negotiation protocol has been successfully used to manage a set of distributed sensors to solve the problem of tracking multiple targets. Specifically, a set of networked sensors cooperatively detect and localize a set of moving targets by taking local measurements and exchanging information. Since a sensor is only capable of measuring the distance from a target to the sensor as well as the speed of the target, in order to estimate the location of the target, multiple sensors have to detect at the same time and combine their measurements. Having measurements from more sensors at the same time or getting the measurements more frequently will produce

a higher quality tracking.

The primary objective of this multiple target tracking problem is to allocate the sensors to the targets so as to maximize the tracking quality. The SPAM protocol is designed for solving this problem with distributed negotiating agents. Each sensor is associated and managed by an agent. Whenever a new target is detected, an agent may also play the role of managing the task of keeping track of the target, i.e., determining which sensors to use for tracking and making schedules for the available sensors. Such an agent is also called a track manager in the protocol. When more than one target enters the system, conflicts on the demand of sensors exist, and track managers may need to negotiate with one another to resolve the conflicts on their local schedules and/or cooperatively produce a global schedule to attempt to maximize the overall tracking quality. The key idea of the SPAM protocol is to select one of the conflicting track managers as a mediator to resolve the conflicts and generate partial solutions for the conflicting managers involved. This conflict resolution process may propagate to multiple negotiation threads. An experimental study showed that the SPAM protocol works very well for this real-time sensor tracking problem. However, the SPAM protocol is too complex to be amenable to a thorough theoretical analysis, and almost all of its important features, such as completeness and convergency, have not been analyzed.

In addition to developing a general experimental approach of analyzing negotiation strategies, another objective of this research is to elucidate the properties of SPAM, including its completeness, convergence, complexity and robustness. To this end, we view the resource (sensor) allocation problem for target tracking as a distributed constraint satisfaction/optimization problem and transform the SPAM protocol into distributed search algorithms. We then characterize many important features of the SPAM protocol by analyzing the derived search algorithms.

This section is organized as follows. We briefly describe the sensor tracking problem and the SPAM negotiation protocol in section 3.1. We formulate the cooperative negotiation problem as a multi-agent constraint problem in section 3.2. In section 3.3, we describe our strategy to characterizing the negotiation as distributed constraint search, and further generalizing and extending the SPAM protocol as two distributed constraint search algorithms. In section 3.4 we apply the derived search algorithms to experimentally analyze some essential features of the negotiation protocol. Finally we conclude and summarize our results in section 3.5.

3.1 Target Tracking and the SPAM Protocol

The application we consider is the tracking of multiple moving targets using a network of loosely coupled sensors. This application has also motivated the SPAM negotiation protocol. We will briefly describe the problem in section 3.1.1 and the protocol in section 3.1.2.

3.1.1 Tracking multiple targets

In this problem, a set of Doppler sensors are scattered with varying orientations throughout a tracking area. Each sensor is able to detect an object within a fixed radius. However, the overall detection area of a sensor is divided into three equal sectors, and the sensor can only operate in one sector at any given time.

The Doppler sensor is only capable of detecting the distance between a target and the sensor by measuring signal amplitude as well as the speed of the target based on signal frequency. In order to track a target, at least three sensors are required to sense at the same time and triangulate the location of the target. Having more sensors tracking at the same time or taking measurements more frequently will produce a higher quality tracking.

The problem can thus be described as follows. Given n sensors and m targets, develop a schedule for all the sensors to track the maximal number of targets as accurately and frequently as possible over a period of time. The complication of the problem stems in at least two factors. First, multiple targets can exist and conflicts on allocating a sensor to multiple targets may be unavoidable, especially when the number of sensors is not sufficient. Second, an agent has partial knowledge of the overall problem but needs to act using local information and information from its neighboring agents.

From utility theoretical point of view, the problem can be considered as a problem of maximizing a utility function of the targets being tracked over a period of time. If the number of sensors allocated to a particular target is less than three, no reliable estimation to the location of the target can be obtained. On the other hand, even though having more than three sensors allocated to a target will give rise to better tracking result, the tracking quality is not linear of the number of sensors used and the quality improvement will be negligibly small after a specific number of sensors. For example, the utility function of tracking a set of m targets can be defined as the total sum of the utility of tracking individual targets, each of which can be defined as $s \times \frac{1}{2}\sqrt{3l-5}$ if l sensors are allocated to a target during a period time s. This utility function could be more complicated if it takes into account the location and the orientation of each individual sensor with respect to the target being tracked.

3.1.2 The SPAM protocol

The SPAM protocol is designed with many considerations on real-time performance and dynamic issues. The protocol is divided into three stages or abstraction levels to cope with three levels of real-time constraints. In the lowest or sensor level, the system is required to respond immediately so that the problem is solved without any information on each local sensor. At the end of this stage, many conflicts over the allocation of the sensors may exist. At the second abstraction level, the problem is solved with knowing the schedule on each local sensor. At the end of this stage, all the local conflicts can be solved, but some non-local conflicts may be introduced due to missing the global information. At the third or the resource level, the conflicting agents negotiate over their local schedules through a mediator who generates partial solutions for all these agents and tries to solve non-local conflicts. As long as conflicts exist in the system, some agent will become a mediator and propagate the negotiation. A brief description of these three stages are given below.

Both simulation and hardware experiments show that the SPAM protocol works very well for the real-time moving target tracking problem. However, since the protocol itself is too complex to analyze, some important features of the protocol are still unclear. We are especially interested in the following properties of the protocol: completeness, time complexity, rate of convergence, and scalability. In order to capture

Algorithm 3 SPAM Protocol Stage 0

```
Evaluate and order the usable resources
Decides an initial objective level
if (have more time) then
   Go to Stage 1
else
   Choose a solution maximizing the local utility
   Bind the solution and exit
end if
```

Algorithm 4 SPAM Protocol Stage 1

```
Collect the local information from all usable resources
Generate the set of local solutions
if (have solution without conflicts) then
   Choose the solution with the maximum local utility
   Bind the solution
else
   Choose a solution which both minimizes the conflicts and maximizes the local utility
   Bind the solution and exit
   if (have more time) then
      Go to Stage 2
   end if
end if
```

these properties, we will transform the protocol into succinct distributed constraint search algorithms. By analyzing these search algorithms, we can characterize many important features of the protocol. Before we launch onto search algorithms, we first describe the constraint formulation of the tracking problem, the topic of the next section.

3.2 Constraint Problems in Cooperative Negotiation

A distributed problem that can be solved by cooperative negotiation normally involves a set of agents, each of which has some tasks to be scheduled using a set of shared resources. Given a set of n agents over m resources, a cooperative negotiation problem can be formally represented as (A, R), where $A = \{a_1, a_2, \cdots, a_n\}$ is the set of n agents and $R = \{s_1, s_2, \cdots, s_m\}$ the set of m resources. An agent, a_i, is represented by a tuple:

$$a_i = (R_i, C_i, w_i),$$

where R_i, $R_i \subseteq R$, is a set of resources that can be used by agent a_i, C_i represents a task to be scheduled by a_i, and w_i is the weight assigned to task C_i.

The task of C_i may also consists of a set of sub-tasks $T_i = \{T_1^{(i)}, T_2^{(i)}, \cdots, T_{t_i}^{(i)}\}$, and (T_i, R_i) constitutes a local sub-problem that can be internally solved by agent a_i. In other words, (T_i, R_i) is local to a_i and is unknown to the other agents. The overall task C_i requires a certain number of resources, and this number, represented by o_i, is called the objective level. Given the set of usable resources R_i and the objective level o_i of the agent a_i, there are $l_i = \binom{|R_i|}{o_i}$ alternative solutions. Then task C_i can be represented

Algorithm 5 SPAM Protocol Stage 2

```
Mediator detects the oscillation by checking the history of negotiation
if (have oscillation) then
   Lower the objective level
end if
Request meta-level information from the conflicting agents
while (have no solution) do
   Generate partial solutions for all these agents
   if (all agents are at their lowest objective level) then
      Choose a solution at the lowest objective level with min-conflicts
      Exit
   else
      Lower the objective level of one agent
   end if
end while
Mediator sends all the partial solutions to the other agents
The other agents evaluate and rank the solutions
Mediator chooses a consistent solution according to the feedback
All the agents bind their solutions
Some agent propagates the negotiation if having conflicts
```

by

$$C_i = \bigvee_{j=1}^{l_i} S_{ij}$$

where S_{ij} represents one possible solution for the agent a_i to the task C_i. And S_i = $\{S_{i1}$,S_{i2},$\cdots$,$S_{il_i}\}$ constitutes the solution space for the sub-problem C_i on agent a_i. Given the objective level o_i, S_{ij} can be further represented by

$$S_{ij} = \bigwedge_{m=1}^{o_i} (r_m^{ij} = i)$$

where $r_m^{ij} \in R_i$. Here $r_m^{ij} = i$ means that the resource r_m^{ij} is allocated to task C_i. Moreover, in order to make C_i true, at lease one solution $sol_i = \bigwedge_{m=1}^{o_i} (r_m^i = i)$, $sol_i \in S_i$, must be true; therefore, all the resource r_m^i should be allocated to C_i. We call sol_i the local sub-solution for the sub-problem C_i on agent a_i.

Here, an agent is a track manager, and a resource corresponds to a sensor in the SPAM protocol. With the restriction that a resource can only be used by one task at any time, each resource should have a consistent allocation or assignment in the local sub-solution of each task C_i, which constitutes a constraint among tasks. Note that each task C_i is distributed among a set of agents. Therefore, we can formulate the cooperative negotiation problem as the following distributed constraint satisfaction problem: *Given a constraint problem* (A, R)*, is there an assignment of resources* $R^+ \subseteq R$ *such that* $\bigwedge_{i=1}^{n} C_i$ *is satisfied?*

As discussed in Section 3.1.1, an agent a_i can have a utility function $U_i : S_i \mapsto \mathbb{R}$ that can be used to discriminate alternative local solutions. Furthermore, the overall goal of the negotiation problem is to find a globally consistent solution with maximal global utility among all the solutions. Therefore, since C_i has

a weight w_i, the global utility can be simply defined as $Util_{total} = \sum_{i=1}^{n} w_i \cdot Util_i$, where $Util_i = U_i(sol_i)$ is the utility of the agent a_i with the local sub-solution sol_i. With the extension, the agent a_i is represented by (R_i, C_i, w_i, U_i), and we can formulate the cooperative negotiation problem as the following constraint optimization problem: *Given a constraint problem* (A, R)*, what is the assignment of resources to agents such that the constraint* $\bigwedge_{i=1}^{n} C_i$ *is true and the utility function* U_{total} *is maximized?*

3.3 Negotiation Protocol as Search Algorithms

The problems solved by cooperative negotiation can be formulated as distributed satisfaction or optimization problems. A negotiation protocol can be viewed as a distributed search process. In this section, we propose to characterize the negotiation protocol by a distributed search algorithm.

3.3.1 Negotiation as distributed search

Negotiation and search are fundamentally different. The former is naturally a multi-agent problem solving method in which information may not be shared among agents. Thus, negotiation is generally harder than search, since a global view of a problem may never be constructed. In such a situation, it may become difficult for an agent to even determine whether the current variable assignments are in a better state than the previous ones. Moreover, a negotiation process may be trapped in an infinite negotiation loop in which all the agents revisit some previously encountered global solutions endlessly. Furthermore, negotiation may have more restrictions than search. For example, time or some other parameters can become a factor when evaluating the results of the negotiation. Thus, the negotiation is normally more complicated in the sense of searching for a solution.

Although negotiation and search are different approaches to problem solving, they both search for assignments to variables of a problem which constitute solutions. Search can also be an ingredient of negotiation. In a negotiation problem, a task of an agent can be viewed as a variable. It can be assigned any values (sub-solutions) in its solution space. The sub-solutions of different variables may conflict with one another due to the inter-agent constraints. When an agent assigns a value to its task, it also needs to make it consistent with the assignments of other agents which is similar to what a normal constraint search algorithm does. An agent's assignment, when communicated to other agents as a proposal in the negotiation, can be rejected by the other agents. Other agents may provide counter-proposals. Thus a negotiation is just a search process, in which the agents try to assign values or revise values to their tasks to satisfy all the inter-agent constraints.

Generally, a negotiation may have two primary goals, to search for consistent solutions and to search for a consistent solution of a maximal utility. The first goal focuses on the conflict resolution, which is a problem solved routinely by constraint satisfaction search algorithms. In this regard, negotiation can thus be viewed as a constraint satisfaction method. The second goal of finding a consistent solution of a maximum utility is simply an optimization problem, a harder search problem. Note that a task of an agent can be a complicated subproblem, and the solution space of the subproblem itself may be large. All agents need to search, cooperatively, to find the best possible global solutions.

In short, negotiation can be viewed as a mechanism for solving distributed constraint satisfaction and constraint optimization problem. Taking this view, we propose to use search algorithms as tools for analyzing negotiation protocols.

3.3.2 SPAM protocol as search algorithms

The original SPAM protocol [68] has many features to handle real-time and dynamics issues. Although these features are very necessary and important to deal with real-world applications, they are hardly amenable to a thorough analysis. It seems to be very difficult to design an abstract model of such a complicated protocol for a theoretical analysis. Therefore, it is difficult to understand, through an analytical approach, some primary properties of the protocol, such as completeness, rate of convergence, complexity and scalability.

The difficulty for a theoretical analysis suggests that an experimental analysis is in demand. Here, we propose to use search algorithms to capture a negotiation protocol so as to characterize the important features of the protocol through analyzing the search algorithms. Once a negotiation protocol is transformed to search algorithms, whenever a theoretical analysis is possible for the search algorithms, such an analysis can also be translated back to the original negotiation protocol.

Another difficulty for analyzing a distributed negotiation protocol comes from the distributed nature of the applications to which the protocol is applied. First, it is usually difficult to set up distributed experiments with a large number of agents and resources using a sufficient number of hardware, while still being able to collect enough accurate experimental data for an evaluation. Second, most experiments in a distributed environment are not repeatable. A distributed negotiation protocol is in essence nondeterministic. There are indeed many factors, for instance the synchronization among agents, that can change experimental results from one run to another.

Therefore, caution must be taken in the experimental analysis. In this section, we will first transform the SPAM protocol to a sequential search algorithm, called sequential SPAM. Here, sequential search does not necessarily mean centralized search. The search process in the sequential SPAM can still be distributed among different agents, but in each step only one agent, chosen arbitrarily, is allowed to change its local values. One run of such a sequential algorithm corresponds to one possible execution of the original negotiation protocol. Introducing sequential execution is expected to have little impact on the effects of the protocol while making the analysis easier. The sequential search algorithm is designed to represent the features of the original protocol as close as possible, so that the results from the search algorithm is expected to shed some light on the original protocol.

We will then modify the SPAM protocol to construct a synchronous search algorithm, called synchronous SPAM. In this version, the agents negotiate in a more tightly cooperative manner. The agents are dynamically ordered during the negotiation. The agent that revises its local assignment earlier has a higher priority. This modification is introduced to make the protocol complete, since the original protocol cannot guarantee the completeness of a negotiation process, as we will see in the next section. Note that completeness is not the same as optimization. If a constraint problem is not satisfiable, the synchronous

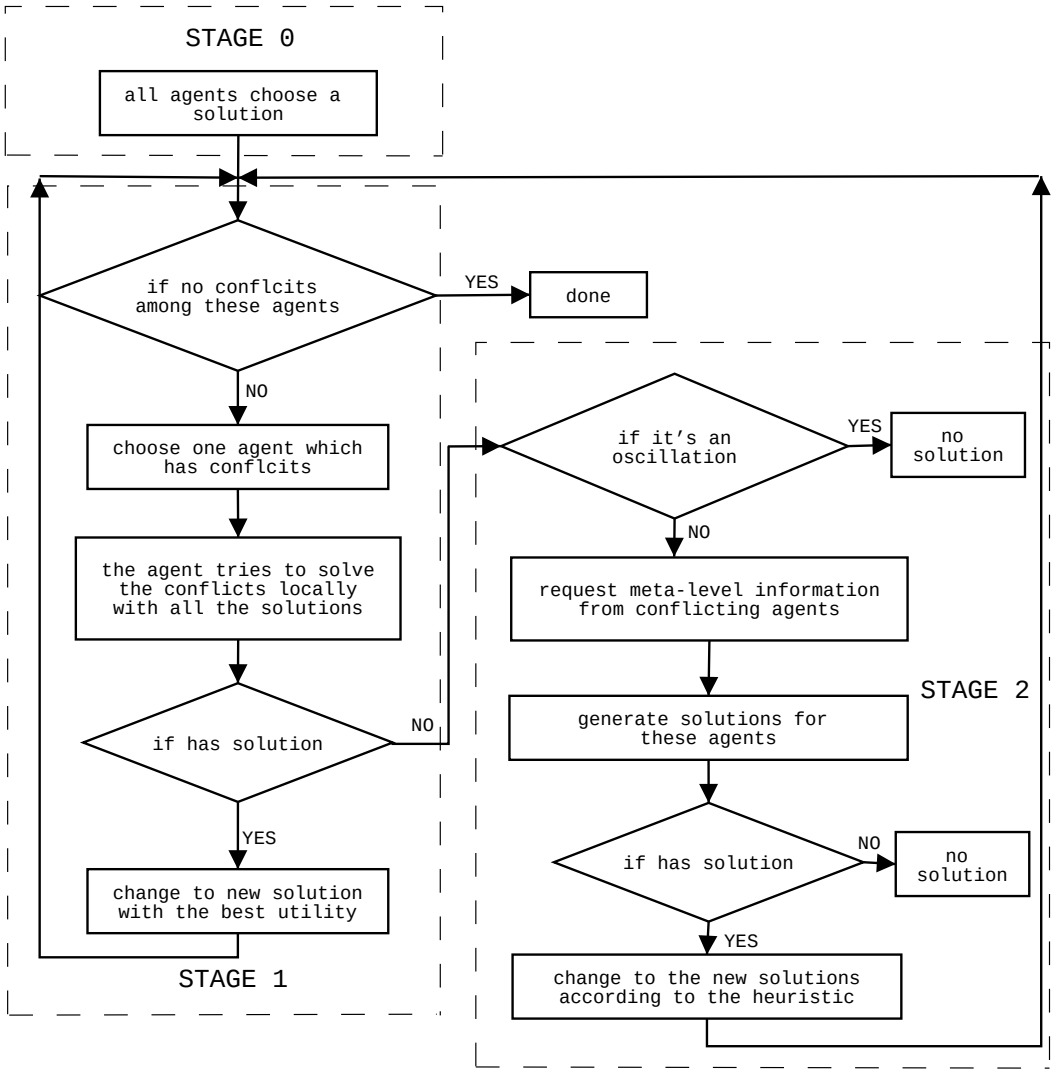


Figure 24: Sequential SPAM

SPAM algorithm is expected to give an answer of 'NO'. Using synchronous SPAM, we are able to evaluate the completeness of the original protocol. Moreover, we also expect to understand how to improve the original protocol by analyzing the synchronous search algorithm.

Sequential SPAM

We now characterize the SPAM protocol as a sequential search algorithm in which only one agent is allowed to change its local values. The protocol is viewed as the following search algorithm, shown in Figure 24.

Similar to the SPAM protocol, this algorithm is divided into three stages. In stage 0, each agent chooses a solution for its local problem without knowing any information about the current schedule of each resource. In stage 1, each agent collects the information of all the resources that can be used. Then one of these agents with conflicts with other agents, chosen randomly, tries to resolve the conflicts locally by searching for a solution in its local solution space. If there exists a local solution that does not conflict

with all the other agents, the agent will change to such a non-conflict solution with the best utility.

If there exists no non-conflict solution, the agent will choose a solution which minimizes the conflicts with the other agents and moves on to stage 2. In other words, this agent becomes a mediator to resolve the conflicts in which it is involved. In stage 2, the agent requests meta-level information from the conflicting agents. It then tries to generate solutions for these agents. If there exist non-conflict solutions, these agents will change to non-conflict solutions based on the heuristic that the most constrained agent chooses a solution first.

If there is still a conflict among agents, a neighboring conflicting agent will attempt to resolve the conflict by propagating the negotiation. To avoid an infinite negotiation loop, each agent records the history of its previous negotiations. If an oscillation is detected, the algorithm terminates or the agent's objective level is reduced (meaning that more resources to be used) to lower the constrainedness of the conflicts. However, as we will see later, this termination condition will make the algorithm lose possible solutions or solutions with higher objective levels. Therefore, the completeness of the algorithm or the protocol can not be guaranteed.

Synchronous SPAM

The synchronous SPAM search algorithm is not exactly the same as the original SPAM protocol. It simulates the original protocol as close as possible and ensures completeness. The algorithm is shown in Figure 25.

In the synchronous SPAM algorithm agents negotiate in a more tightly cooperative manner. The algorithm has additional features, such as priority and commitment, to guarantee the completeness. In this algorithm, each agent has a tentative initial solution for its local problem. The tentative solution is revised when the agent commits its solution. All the committed sub-solutions constitute a partial solution to the overall problem. The revised solutions must satisfy all the constraints with the sub-solutions. If there exist multiple solutions, a min-conflict heuristic [76] is used to minimize as many conflicts with tentative solutions as possible. If no solution exists, the partial solutions will be added as a new constraint, and all the committed solutions will be uncommitted and become tentative solutions again. Here the idea is borrowed from weak-commitment search algorithm [106] which has proved to be more efficient than backtracking algorithms in many cases. After the partial solution is uncommitted, the agent will request meta-level information from the other conflicting agents and try to generate solutions for all of these agents. If no solution exists, the whole problem will have no solution at the current objective level. As a result, some agents must lower their objective levels, i.e., reduce some constraints of their local problems. Otherwise, these agents will just commit the solutions. Note that when there is no solution, the algorithm will not try to minimize the number of violated constraints. Thus this algorithm is for solving constraint satisfaction not optimization.

As we will see in the next section, this algorithm can guarantee completeness. It can also be easily modified into an asynchronous version in which multiple agents can search in parallel but still guarantee the completeness.

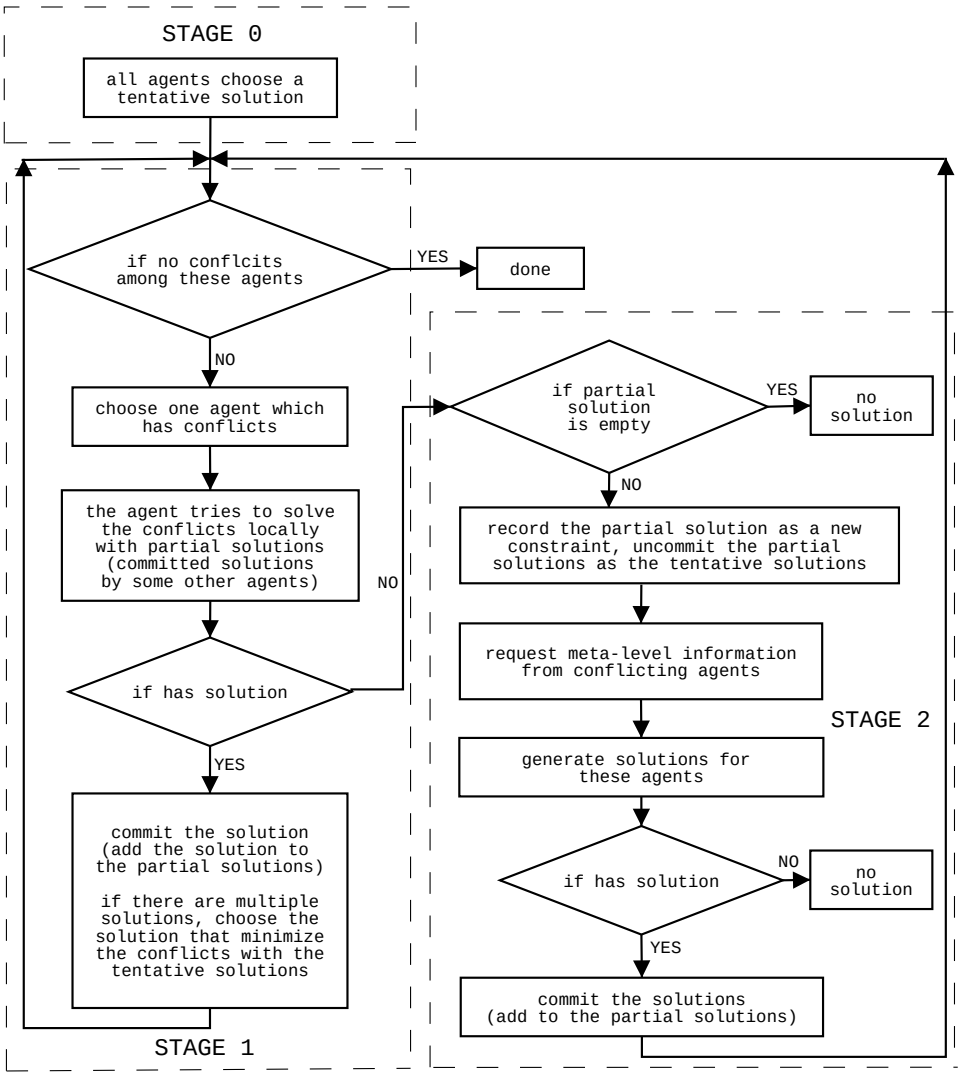


Figure 25: Synchronous SPAM

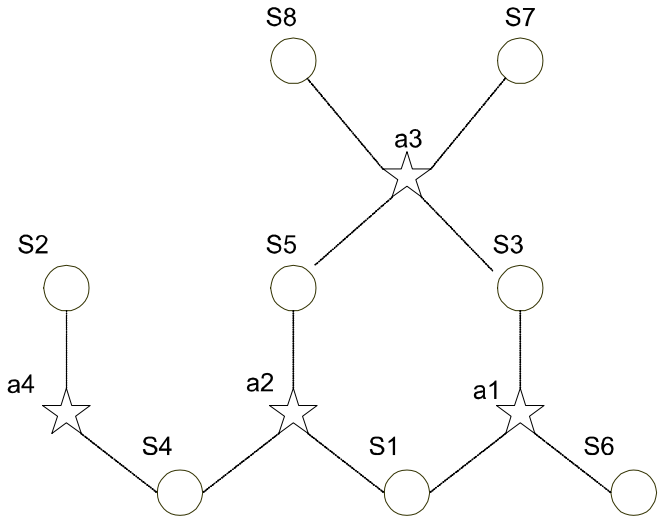
3.4 Experimental Analysis and Results

To reveal the properties of the SPAM protocol, we now analyze the completeness, convergence rate, complexity and scalability of sequential and synchronous SPAM algorithms. Here, the completeness of a protocol is the capability of finding a solution if one exists; the convergence rate concerns the number of negotiation steps required to reach a solution; the complexity measures the total number of steps taken before a protocol terminates; and the scalability considers how the properties of a protocol change as the size of a system increases.

3.4.1 Completeness

The synchronous SPAM protocol can be proved complete. Since the algorithm records the abandoned partial solutions as new constraints, the algorithm will not create the same partial solution twice. Therefore, the completeness of the protocol is guaranteed, because there are a finite number of partial solutions to be

enumerated. The worst-case time complexity of this protocol is obviously exponential in the number of agents. Assuming that there are n agents, and each agent has a solution space of size S, the worse case time complexity will be $O(S^n)$. Since the problem itself is NP-complete, this result seems inevitable. The worst case space complexity of this protocol is also exponential in the number of agents since if there is no solution for the whole problem, all the partial solutions will be added as new constraints. However, unlike most of the tree search algorithms, the synchronous SPAM changes the search order flexibly, which makes it more efficient since it avoids exhaustively searching all the bad solutions when previous values are set wrong.



Problem Matrix

	s1	s2	s3	s4	s5	s6	s7	s8
a1	1		1			1		
a2	1			1	1			
a3			1		1		1	1
a4		1		1				

Initial Assignment

	s1	s2	s3	s4	s5	s6	s7	s8
a1	1					1		
a2	1				1			
a3			1				1	
a4		1		1				

conflicts

Figure 26: An example for incompleteness

The sequential SPAM protocol is not complete. This is simply because that the agents in this algorithm do not follow any ordering of assignments during the search. This will cause some agents to cycle through some sub-solutions and fall into an infinite search or negotiation loop. In principle, the algorithm is similar to a local search and can step on a loop in search space or be trapped by local minima. To avoid such an infinite loop or local minimum, the protocol simply gives up the current search by lowering the current objective level if an oscillation is detected. This makes the algorithm incomplete to find some possible solutions.

Figure 26 gives a simple example of incompleteness where four agents (track managers) try to uti-

lize eight resources (sensors) for target tracking. The problem has a solution which gives each agent a maximum objective level of two. The synchronous SPAM algorithm can find the solution after several steps given the initial assignment shown in the Figure 26, while the sequential SPAM fails to find such a solution, due to oscillations in search processes.

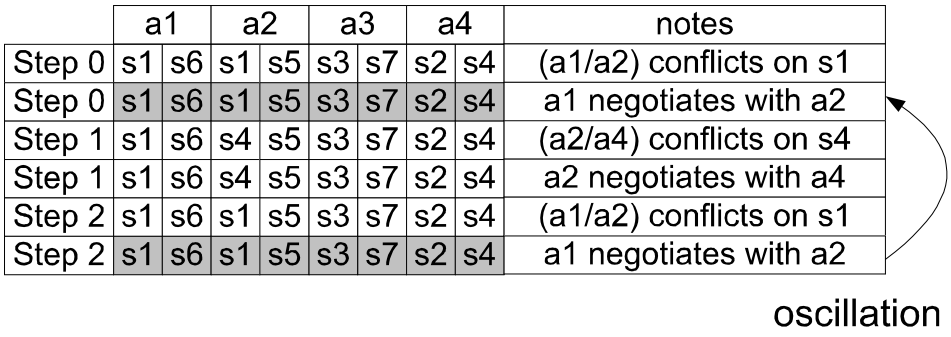


Figure 27: Execution of sequential SPAM on the example

Figure 27 illustrates an example of execution steps of sequential SPAM. For the initial assignments (after stage 0), a_1 and a_2 have conflicts on the resource s_1. a_1 searches locally but finds that there is no local solution to resolve the conflicts (after stage 1). It then goes into stage 2 to negotiate with a_2. It finally finds a partial solution (assign (s_4, s_5) to a_2 and (s_1, s_6) to a_1) for both a_1 and a_2 which resolves conflicts. So in step 2, new values are assigned. Since a_2 and a_4 still have conflicts on s_4, a_2 propagates the negotiation. But note that a_4 has only two resources available, the only partial solution for a_2 and a_4 is to assign (s_2, s_4) to a_4 and (s_1, s_5) to a_2. So in step 2, new values are assigned. Now, if a_1 is to propagate the negotiation, it will find that the same situation in step 2 as in step 0. An oscillation is detected which makes the algorithm exit without finding the possible solution.

Although sequential SPAM is not complete, our experiments show that the possibility for the algorithm to be complete is very high, especially in under-constrained situations. The experiment is set up with 10 agents and 20 agents. The number of resources are 30 and 60 respectively. Each task has a fixed objective level of 3, which is the best possible in both situations. The availability of resources to agents varies from 0.1 to 0.9. The availability here simply means the probability that any resource can be used by an agent. For example, in our experimental setting of 10 agents and 30 resources, when availability equals 0.5, on average each agent will have 15 resources to use. This implies that when the availability increases, the constrainedness of the problem decreases. Given a set of agents, a set of resources, and the availability p, each problem instance in our experiment is generated by randomly adding an edge between an agent and a resource with the probability p. An edge between an agent and a resource simply means that the resource is available to the agent.

Figure 28 shows the ratio of the problems solved by sequential and synchronous SPAM algorithms over 10,000 problem instances, with different resource availability. Since the synchronous SPAM algorithm is complete, the results of synchronous SPAM give a baseline for sequential SPAM. The result for the 10 agents case shows that when the resource availability is bigger than 0.4, most of the problems are solvable;

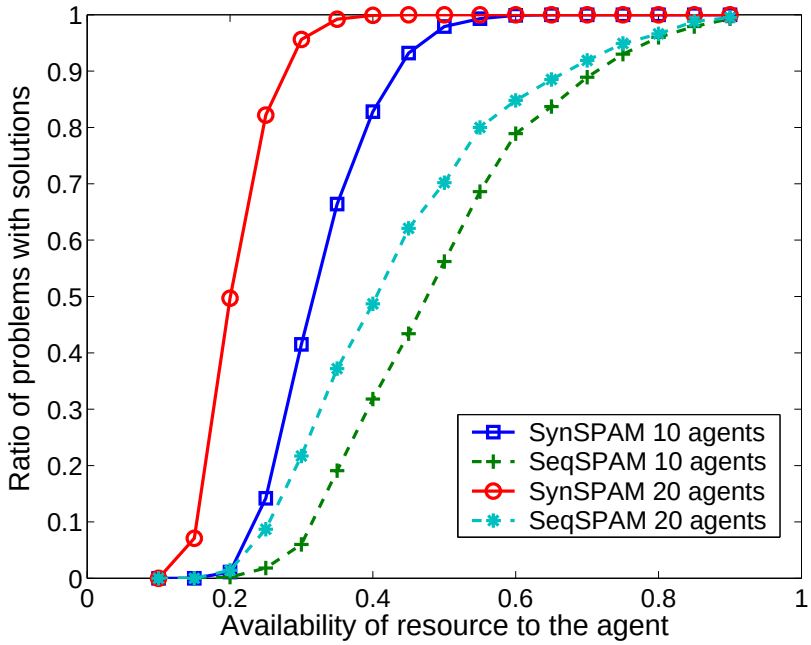


Figure 28: Number of problems solved by each algorithm

whereas when the resource availability is less than 0.2, most of the problem are unsolvable. As with other CSPs, it is expected that the hardest instances of this negotiation problem are more likely to occur when the resource availability is between 0.2 and 0.4, where around half of the problems are solvable. Notice that when the problem size increases to 20 agents, the transition from the region with most unsolvable problems to the region with most solvable problems is even sharper. We suspect that a phase transition may exist in the negotiation problem.

Figure 29 shows the ratio of completeness of the sequential SPAM algorithm as the resource availability of the problem increases or the constrainedness of the problem decreases. This figure comes from the same results plotted in Figure 28. Each data point equals to the number of problems solved by the sequential SPAM algorithm divided by the number of problems solved by the synchronous SPAM algorithm. Since synchronous SPAM is a complete algorithm, the ratio directly reflects the ratio of completeness of sequential SPAM. The result shows that as the resource availability increases, the completeness of sequential SPAM increases as well. For the 20 agents problem, when the availability is greater than 0.4, the completeness ratio of sequential SPAM is more than half. When the resource availability increases beyond 0.6, more than 80% of the solvable problem instances are solved by the sequential SPAM algorithm. This simply indicates that the original SPAM protocol is well suited for under-constrained problems, having a very high possibility to be complete.

3.4.2 Time complexity

The sequential SPAM algorithm sacrifices completeness for computation time, as to be verified by experiments. Again, we run both sequential and synchronous SPAM algorithms on 10,000 problem instances

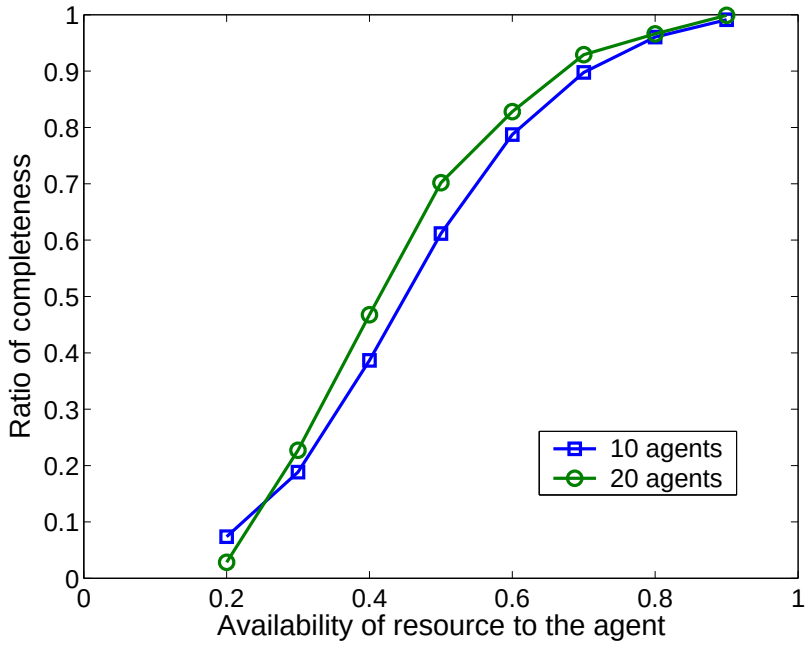


Figure 29: The rate of completeness for sequential SPAM

with different resource availabilities. The problem settings are still 10 agents with 30 resources. Figure 30 plots the average total CPU time of each algorithm for these 10,000 instances in second. Both experiments were on a linux machine with 756 MB memory and an AMD 1.4GHZ processor.

The result of the synchronous SPAM algorithm shows a phenomenon similar to that of phase transitions. It takes more CPU times on problems with resource availability at 0.3 and 0.4 than the problems in the other resource availability. The results here are consistent with the results in Figure 28 which shows that the problems around 0.3 and 0.4 availability are located in the middle of a phase transition on solubility, and thus are harder to determine quickly if they are solvable or not.

However, the result of the sequential SPAM algorithm does not show any phase-transition phenomenon. Its CPU time smoothly increases with the resource availability. Under all the availability settings we considered, the sequential SPAM algorithm takes less CPU time than the synchronous SPAM algorithm. One explanation is that the sequential algorithm is able to give up searching for a solution sooner on hard problems than the synchronous algorithm so that the former finishes faster than the latter. Another, minor reason is that the sequential SPAM algorithm has a lower overhead in each step.

One simple implication of these results is that there exists a tradeoff between the completeness and time complexity of a protocol. Adding additional features such as those we introduced in the synchronous algorithm may make a protocol complete, but meanwhile may decrease the time performance of the protocol. For many soft-constrained problems in practice where the completeness is not crucial, using incomplete algorithms or protocols such as the SPAM protocol seems to be the right choice.

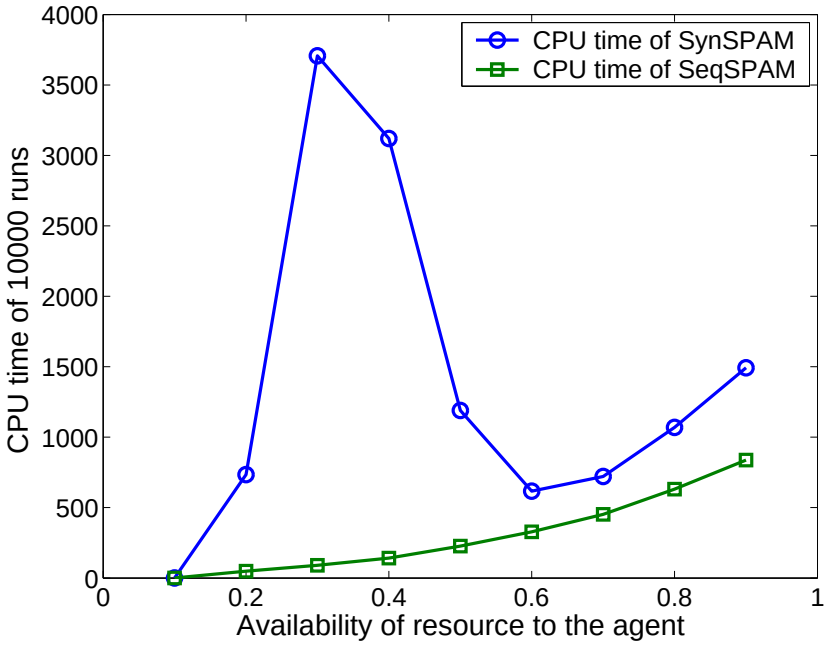


Figure 30: The CPU time of each algorithm

3.4.3 Convergency and performance

The sequential and synchronous algorithms cannot guarantee to always turn the current state into a better one after each step of negotiation. For asynchronous SPAM, during the stage 1, when an agent commits a local solution that has no conflict with partial solutions, this new solution may introduce more conflicts with tentative solutions than the conflicts it can reduce. Consequently, the overall conflicts are increased after one step of negotiation. Similar example can also be found in stage 2 of the sequential SPAM algorithm. However, both algorithms are able to terminate after a finite number of steps. Therefore, both algorithms will finally converge to some specific values. For the synchronous SPAM algorithm, this value is 0 if the problem is solvable as the algorithm is complete, while for the sequential SPAM algorithm, this value may not necessarily be 0.

Here, we are particularly interested in the rate of convergence of these two algorithms, which basically measures how fast the protocols improve solution quality. Figure 31 shows the experimental results with 10 agents and 30 resources averaging over 10,000 problem instances, indicating that the convergency speed of the sequential algorithm is normally better than that of the synchronous algorithm. This result is not surprising. If we compare the two algorithms, we will find that in each step, the synchronous SPAM algorithm tries to resolve the conflicts with only partial solutions, whereas the sequential SPAM tries to resolve the conflicts with all the solutions. This means that in each step, the sequential SPAM is likely to resolve more conflicts than the synchronous SPAM, so that its convergency speed may be faster. Figure 32 also shows the results on some larger problems with 20 agents and 30 agents.

Based on these results, we find the following interesting facts. Despite its incompleteness, when the sequential SPAM algorithm terminates, the final solutions are almost always near optimal. Specifically, the

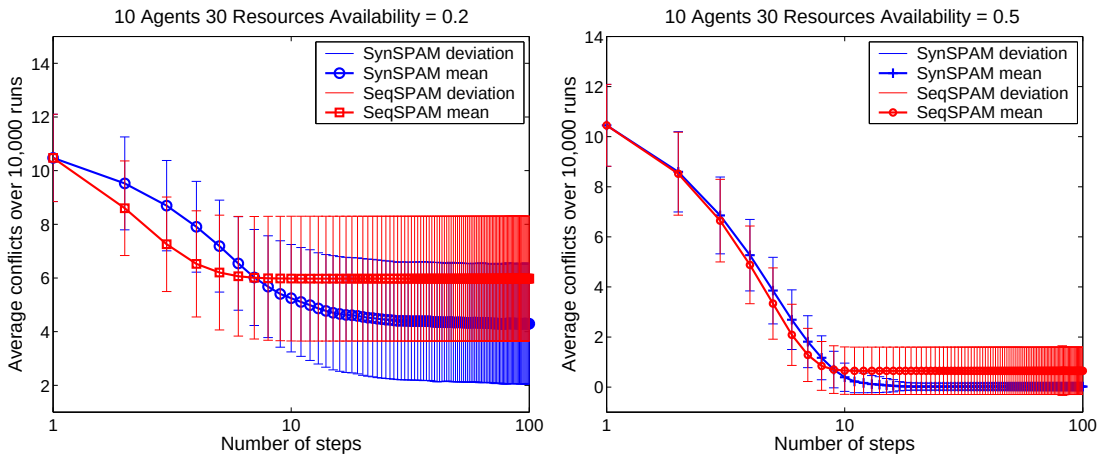


Figure 31: The convergency speed of two algorithms

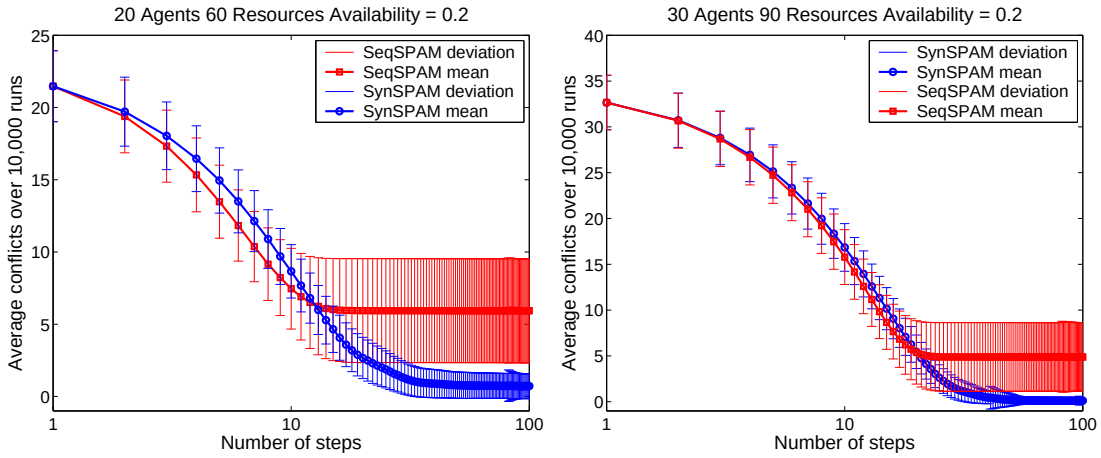


Figure 32: The convergency speed of two algorithms

average solutions from sequential SPAM are only a few conflicts more than that of synchronous SPAM, especially in the under-constrained situations. This is summarized in Table 2, which shows the performance results of two algorithms on tracking problems with different problem sizes and resource availabilities. Each data point in the table shows the average number of conflicts unsolved after 100 step execution of the algorithms, averaged over 10,000 random problem instances. Based on the table, most of the results of the sequential algorithm are comparative to the synchronous algorithm, especially in the under-constrained cases. For 10-agent problems, the biggest performance gap occurs when the resource availability is equal to 0.3, which is located near the middle of a phase transition on solubility. For 20 agent problems, the phase transition point shifts to the availability of 0.2, and correspondingly the biggest performance gap shifts as well. In all other regions apart from phase transition, the performance differences between the two algorithms are trivial.

In light of all the above results and the results on the completeness, we can conclude that for many real-time applications, it is reasonable to give up the total completeness of SPAM protocol in favor of a

Table 2: Mean Conflicts Over 10,000 problem instances After 100 Steps

size	*algo.*	*resource availability*					
		0.2	*0.3*	*0.4*	*0.5*	*0.6*	*0.8*
10 agents	*Syn. SPAM*	4.2956	0.8491	0.1761	0.0173	0.0016	0
	Seq. SPAM	5.9764	3.0456	1.3994	0.6950	0.2799	0.0409
20 agents	*Syn. SPAM*	0.7201	0.0377	0.0016	0	0	0
	Seq. SPAM	5.9953	1.9827	0.8805	0.4009	0.1965	0.0342

faster convergency to a good enough solution.

3.4.4 Scalability

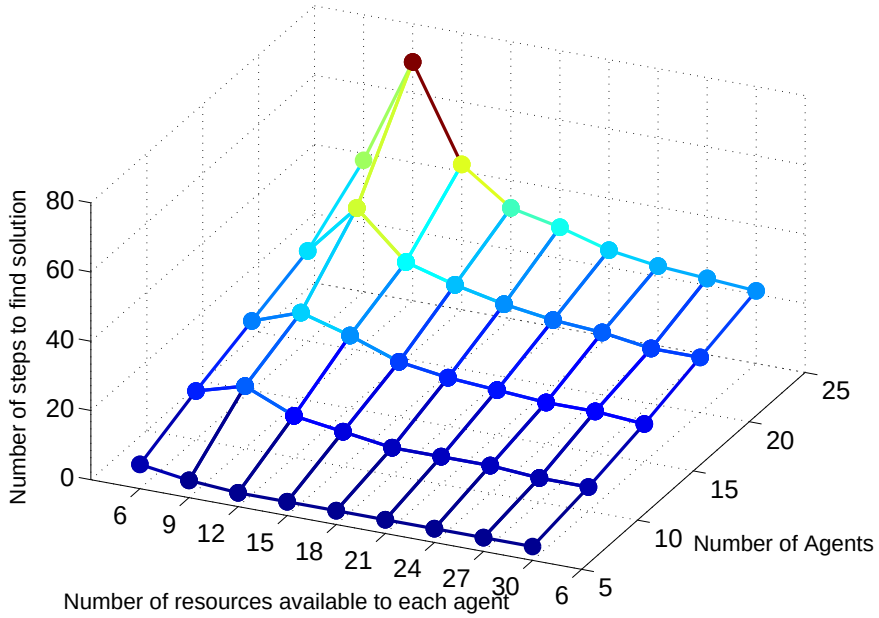


Figure 33: Synchronous SPAM scalability

Another important aspect of performance analysis of a protocol is the study of how the performance varies as problem size increases, i.e., how the protocol scales. This feature is particularly important in applications such as sensor networks in ANTs domains, since such systems can easily have hundreds or thousands of sensors. A good scalability means that system performance does not degrade disproportionally with the size of the system or does not degenerate at all when the number of components increases. Here we are interested in answering the following question: will the number of negotiation steps that SPAM protocol takes to find a solution dramatically increase as the number of agents increases?

Figure 33 shows the experimental results for the synchronous SPAM protocol, in which the number of agents varies from 5 to 25, and the average number of resources available to each agent varies from 6 to 30. The total number of resources is three times of the number of agents in each of 1,000 random problem

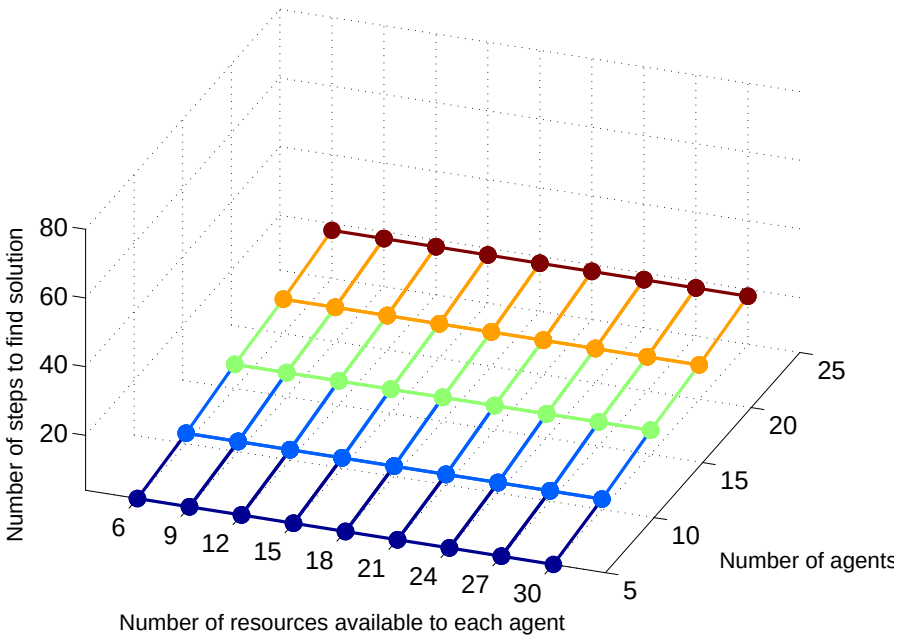


Figure 34: Sequential SPAM scalability

instances. In other words, the objective level for each agent is set to 3. Note that for the 5-agent problems, the maximum number of resources available to each agent is 15. We simply extend the data on 15 to 30 in the 5-agent case to make the results easy to show. In Figure 33, each data point represents the average number of steps over the solvable problems of 1,000 runs. From the results, we can see that the scalability of synchronous SPAM is super linear. Particularly around the phase transition area, the performance of the protocol substantially degrades.

Figure 34 shows the experimental results on the sequential SPAM protocol with the same experiment setup as above. Again each data point represents the average result over the 1,000 solvable problem instances. Comparing to the results in Figure 33, Figure 34 shows that the scalability of sequential SPAM is better than the scalability of synchronous SPAM. However, based on these results, we are unable to affirm that the scalability of sequential SPAM is linear. Nevertheless, the results indicate that the sequential SPAM algorithm does not seems to have a substantial degrading region around the phase-transition area similar to the synchronous SPAM algorithm. We should also note that at each point in Figure 34, fewer problems are completely solved than the corresponding point in Figure 33.

3.4.5 Summary

In this section, we experimentally analyze the properties and performance of two distributed search algorithms derived from a recently developed cooperative negotiation protocol for resource allocation in networks of distributed sensors. The sequential version simulates the protocol as close as possible, while the synchronous version adds some additional features to ensure the completeness. The experimental results on these algorithms help us understand the properties of the original protocol. Specifically, our results

show that the SPAM protocol is not complete in terms of finding a solution. However it has a high probability of finding a solution if one exists, especially in under-constrained situations. The SPAM protocol is able to give up sooner on the hard problems so as to finish faster than the complete protocols. The protocol converges well in the sense that it can find good low-conflict solutions within a small number of steps. The solution quality is comparative to the complete protocol in most of the cases. The protocol seems to scale very well as the number of agents and the number of resources increase.

With all these results above and the fact that the SPAM protocol is normally used in a dynamic environment, it is reasonable to give up the completeness of the protocol in favor of computational complexity, convergency speed, and scalability. Note that in a dynamic environment, even if the protocol finally finds the optimal solution with a complete search, the problem could have already changed and the optimal solution may no longer be relevant. Therefore, making the right tradeoff between solution quality and computational time is critical, and the SPAM negotiation protocol seems to make such a tradeoff very well for real-time moving target tracking.

3.5 Conclusions and Discussions

Motivated to understand the properties of a recently developed cooperative negotiation protocol, we proposed an approach to analyzing distributed negotiation methods. In this approach, we view and formulate a negotiation protocol as a distributed search, and then experimentally investigate the properties and performance of the search algorithms to help draw conclusions on the original protocol. We demonstrated this approach on the Scalable Protocol for Anytime Multi-level (SPAM) negotiation protocol for allocating resources among a set of cooperative distributed sensors. In addition to the contribution on a negotiation protocol itself, we substantially extended the well known notion of distributed AI as distributed search [64]. We not only viewed a distributed cooperative negotiation as a distributed search, we also proposed to use search as a simulation tool to analyze negotiation protocols. As we demonstrated in this section, this approach can overcome many difficulties inherent to a negotiation protocol that is hard to investigate analytically. We believe that this approach is general and can be carried over to analyzing other distributed problems and strategies.

In reflecting on the experimental results on the performance of the SPAM negotiation protocol obtained in this section, it seems that high-quality global solutions to a distributed problem can be achieved without a global control but with a negotiation protocol in which agents rely on information restricted to their small neighborhoods. With a small locality of information sources, anytime performance of a system can also be significantly improved. An important lesson we have learnt from the results of this section is that propagating information among neighboring agents and using information in a small neighborhood vicinity may be a good general strategy for distributed problem solving.

4 Phase Transitions and Backbones of the Asymmetric Traveling Salesman

Phase transitions of combinatorial problems and threshold behavior similar to phase transitions in combinatorial algorithms have drawn much attention in recent years [37, 44, 70]. Introduced first in the so-called spin glass theory [75] in physics, phase transition refers to such a phenomenon of a system in which some global properties change rapidly and dramatically when a control parameter goes across a critical value. A daily-life example of phase transitions is water changing from ice (solid phase) to water (liquid phase) to steam (gas phase) when the temperature increases. It has been shown that many combinatorial decision problems have phase transitions, including Boolean satisfiability [14, 77, 43, 88, 78], graph coloring [14], and the symmetric Traveling Salesman (deciding the existence of a complete tour of vising a given set of cities with a cost less than a specified value) [33]. Phase transition can be used to characterize typical-case properties of difficult combinatorial problems [37, 70]. The hardest problem instances of most decision problems appear very often at the points of phase transitions. Therefore, phase transitions have been used to help generate the hardest problem instances for testing and comparing algorithms for decision problems [1, 14, 77].

Another important and useful concept for characterizing combinatorial problems is that of backbone [61, 78]. A backbone variable refers to such a variable that has a fixed value among all optimal solutions of a problem; and all such backbone variables are collectively referred to as the backbone of the problem. If a problem has a backbone variable, an algorithm will not find a solution to the problem until the backbone variable is set to its correct value. Therefore, the fraction of backbone, the percentage of variables being in the backbone, reflects the constrainedness of the problem and directly affects an algorithm searching for a solution. The larger a backbone, the more tightly constrained the problem becomes. As a result it is more likely for an algorithm to set a backbone variable to a wrong value, which may consequently require a large amount of computation to recover from such a mistake.

However, the research on the phase transitions and (particularly) backbones of *optimization problems* is limited, which is in sharp contrast with the numerous studies of the phase transitions and backbones of decision problems, represented by Boolean satisfiability (e.g., [14, 77, 43, 88, 78]). An early work on the symmetric TSP introduced the concept of backbones [61]. However, it has left the question whether there exists a phase transition of the TSP, the optimization version of the problem to be specific, open since 1985. One of the best (rigorous) phase-transition results was obtained on number partitioning [10], an optimization problem. However, the phase transition analyzed in [10], also experimentally in [34, 35], is the existence of a perfect partition for a given set of integers, which is in essence a decision problem. In addition, [34, 35] also studied the phase transitions of the size of optimal 2-way partition. The relationship between the phase transitions of satisfiability, a decision problem, and maximum satisfiability, an optimization problem, was studied in [108]. It was experimentally shown that the backbone of maximum Boolean satisfiability also exhibits phase transitions, emerging from nonexistence to almost a full scale abruptly and dramatically. In addition, the relationship between backbones and average-case algorithmic complexity has also been considered [90].

In this section, we investigate the phase transitions of the asymmetric Traveling Salesman Problem.

Our results answer the long-standing open question of [61] based on the more general form of the problem. The Traveling Salesman Problem (TSP) [38, 63] is an archetypical combinatorial optimization problem and one of the first NP-hard problems studied [52]. Many concepts, such as backbone [61], and general algorithms, such as branch-and-bound [66], local search [65] and simulated annealing [86] were first introduced and studied using the TSP. The problem is also very often a touchstone for combinatorial algorithms. Furthermore, the fact that many real-world problems, such as scheduling and routing, can be cast as TSPs has made the problem of practical importance. In this section, we consider the asymmetric TSP (ATSP), where a distance from one city to another may not be necessarily the same as the distance on the reverse direction. The ATSP is more general and most ATSPs are more difficult to solve than their symmetric counterparts [51].

Specifically, using uniformly random problem instances of up to 1,500 cities, we empirically reveal that the average optimal tour length, the accuracy of the most effective lower-bound function for the problem (the assignment problem [69]), and the backbone of the ATSP undergo sharp phase transitions. The control parameter is the precision of intercity distances which is typically represented by the maximum number of digits for the distances. Note that these results are algorithm independent and are properties of the problem. Furthermore, we show that the average computational cost of the well-known branch-and-bound subtour elimination algorithm [4, 7, 92] for the ATSP exhibits a phase-transition or threshold behavior in which the computational cost grows abruptly and dramatically from easy to difficult as the distance precision increases. Our results lead to a practical guidance on how to generate large, difficult random problem instances for the purpose of algorithm comparison.

It is worthwhile to mention that besides the results in [61] there are two additional pieces of early empirical work related to the phase transitions of the Traveling Salesman. The research in [113] investigated the effects of two different distance distributions on the average complexity of the subtour elimination algorithm for the asymmetric TSP. The main result is that the average complexity of the algorithm is controlled by the number of distinct distances of a random asymmetric TSP. We will extend this result further in Section 4.5. However, we need to caution that these results are algorithm specific, which may not necessarily reflect intrinsic features of the underlying problem. The research in [33] studied the decision version of the symmetric TSP. Specifically, it analyzed the probability that a tour whose length is less than a specific value exists for a given random symmetric euclidean TSP, showing that the probability has a one-to-zero phase transition as the length of the targeting tour increases. Note that this phase-transition result does not address the open question of [61] which is on the optimization version of the problem. The experimental results of [33] also showed that the computational cost of a branch-and-bound algorithm, which unfortunately was not specified in this section, exhibits an easy-hard-easy pattern.

This section is organized as follows. In Section 4.1, we describe the ATSP and a related problem called assignment problem. We then investigate the parameter that controls phase transitions in Section 4.2, and study various phase transitions of the ATSP in Section 4.3. In Section 4.5, we describe the well-known subtour elimination algorithm for the ATSP, and analyze the threshold behavior of this algorithm. We discuss related work in Section 4.6 and finally conclude in Section 4.7.

4.1 The Asymmetric Traveling Salesman and Assignment Problem

Given n cities and the distance or cost between each pair of cities, the *Traveling Salesman Problem* (TSP) is to find a minimum-cost tour that visits each city once and returns to the starting city [38, 63]. When the distance from city i to city j is the same as the distance from j to i, the problem is the symmetric TSP (STSP). If the distance from city i to city j is not necessarily equal to the reverse distance, the problem is the asymmetric TSP (ATSP). The ATSP is more difficult than the STSP, with respect to both optimization and approximation [51]. The TSPs are important NP-hard problems [32, 52] and have many practical applications. Many difficult combinatorial optimization problems, such as vehicle routing, workshop scheduling and computer wiring, can be formulated and solved as the TSPs [38, 63].

The ATSP can be formulated as an integer linear programming problem. Let V be the set of n cities, A the set of all pairs of cities, and $D = (d_{ij})$ the distance or cost matrix specifying the distance of each pair of cities. The following integer linear programming formulation of the ATSP is well known:

$$ATSP(D) = \min \sum_{i,j} d_{ij} x_{ij}, \tag{1}$$

subject to

$$\sum_{i \in V} x_{ij} = 1, \qquad \forall j \in V; \tag{2}$$

$$\sum_{j \in V} x_{ij} = 1, \qquad \forall i \in V; \tag{3}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \quad \forall S \subset V, S \neq \emptyset; \tag{4}$$

$$x_{ij} \geq 0, \qquad \forall i, j \in V \tag{5}$$

where variables x_{ij} take values zero or one, and $x_{ij} = 1$ if and only if arc (i, j) is in the optimal tour, for i and j in V. Constraints (2) and (3) restrict the in-degree and out-degree of each city to be one, respectively. Constraints (4) impose the *subtour elimination* constraints so that only complete tours are allowed.

The ATSP is closely related to the *assignment problem* (AP) [69], which is to assign to each city i another city j, with the distance from i to j as the cost of the assignment, such that the total cost of all assignments is minimized. The AP is a relaxation of the ATSP in which the assignments need not form a complete tour. In other words, by removing the subtour elimination constraints (4) from the above representation, we have an integer linear programming formulation of the AP. Therefore, the AP cost is a lower bound on the ATSP tour cost. If the AP solution happens to be a complete tour, it is also a solution to the ATSP. While the ATSP is NP-hard, the AP can be solved in polynomial time, in $O(n^3)$ to be precise [69].

4.2 The Control Parameter

Consider two cases of the ATSP, one with all the intercity distances being the same and the other with every intercity distance being distinct. In the first case, every complete tour going through all n cities is

an optimal tour or a solution to the ATSP. There is no backbone variable since removing one edge from an optimal solution will not prevent finding another optimal solution. The ATSP in this case is easy; finding an optimal solution does not require any search at all. In addition, the cost of the optimal solution is also a constant, which is n times of the intercity distance. In the second case where all distances are distinct, every complete tour covering all n cities has a high probability to have a distinct cost. Therefore, an arc in the optimal solution is almost surely a backbone variable and removing it may destroy the optimal solution. In addition, it is expected to be difficult to find such an optimal solution among a large number of suboptimal solutions in this case.

Therefore, there are significant differences between the above two extreme cases. One of the most important differences is the number of distinct distances in the distance matrix D. More precisely, many important characteristics of the random ATSP, including the size of backbone and complexity, are determined by the fraction of distinct distances among all distances. We denote the fraction of distinct distances in distance matrix D as ρ. We are particularly interested in determining how ρ affects the characteristics of the ATSP when it gradually increases from zero, when all distances are the same, to one, when all distances are distinct.

In practice, however, we do not directly control the number or the fraction of distinct distances in matrix D. Besides the actual structures of the "layouts" of the cities, the precision of the distances affects the number of distinct distances. The precision of a number is usually represented by the maximal number of digits allowed for the number. This is even more so when we use a digital computer to solve the ATSP, which typically has 32 bits (or 4 bytes) for integers or 64 bits (or 8 bytes) for double precision real numbers. As a result, the number of digits for distances is naturally a good choice for the control parameter.

The effect of a given number of digits on the fraction of distinct distances in distance matrix D is relative to the problem size n. Consider a matrix D with distances uniformly chosen from integers $\{0, 1, 2, \cdots, R-1\}$, where the range R is determined by the number of digits b. For a fixed number of digits b, the fraction of distinct distances of a larger matrix D is obviously smaller than that of a smaller D. Therefore, the control parameter for the fraction ρ of distinct distances of D must be a function of the number of digits b and the number of cities n, which we denote as $\rho(n, b)$.

To find the control parameter, consider the number of distinct distances in D for a given integer range R. The problem of finding the number of distinct distances is equivalent to the following bin-ball problem. We are given M balls and R bins, and asked to place the balls into the bins. Each ball is independently put into one of the bins with an equal probability. We are interested in the fraction of bins that are not empty after all the placements. Here, for asymmetric TSP $M = n^2 - n$ balls correspond to the total number of nondiagonal distances of matrix D, and R bins represent the possible integers to choose from. Since each ball (distance) is thrown independently and uniformly into one of the R bins (integers), the probability that one bin is not empty after throwing M balls is $1 - (1 - 1/R)^M$. The expected number of occupied bins (distinct distances) is simply $R\left(1 - (1 - 1/R)^M\right)$. Thus, the expected fraction of distinct distances in matrix D is

$$E[\rho(n, b)] = \frac{R\left(1 - (1 - 1/R)^M\right)}{M} \tag{6}$$

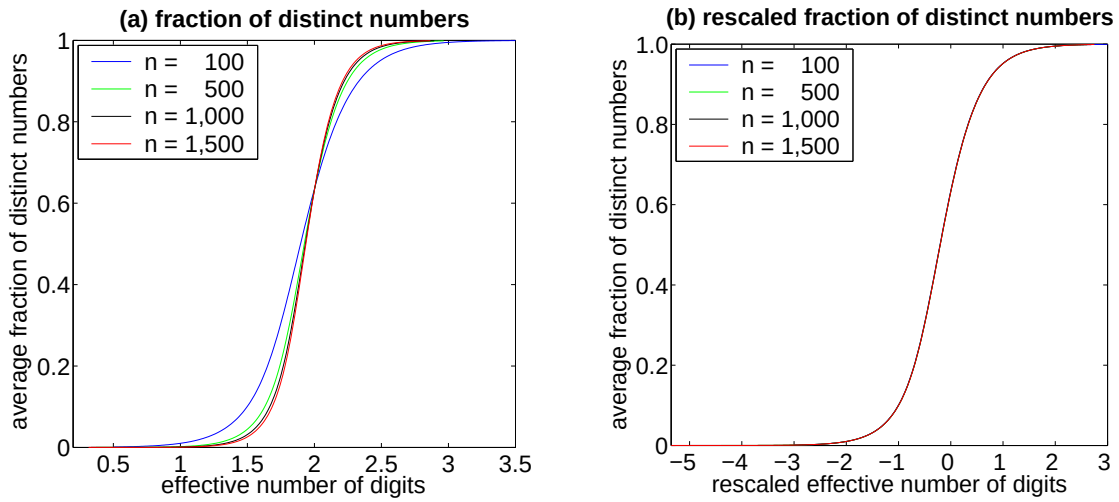


Figure 35: (a) Average fraction of distinct distances in matrix D, $\rho(n, b)$, controled by the effective number of digits, $\beta = b \log_{10}^{-1}(n)$, for $n = 100, 500, 1000$ and 1500. (b) Average $\rho(n, b)$ after finite-size scaling, with scaling factor $(\beta - \beta_c) \log_{10}(n)$, where $\beta_c = 2$.

Note that if M or n is fixed, $E[\rho(n, b)] \to 1$ as $R \to \infty$, since in this case the expectation of the number of distinct distances approaches M. On the other hand, when R is fixed, $E[\rho(n, b)] \to 0$ when M or n goes to infinity, since all of a finite number of R bins will be occupied by an infinite number of balls in this case.

Following convention in practice, we use decimal values for distances. Thus $R = 10^b$, where b is the number of digits for distances. It turns out that if we plot $\rho(n, b)$ against $b/\log_{10}(n)$, it will have relatively the same scale for different problem sizes n. This is shown in Figure 35(a). This means that the scaling function for the effective number of digits is $f(n) = \log_{10}^{-1}(n)$. Function $b/\log_{10}(n)$ is thus the *effective number of digits* that controls the fraction of distinct distances in matrix D, which we denote as $\beta(n, b)$. This also means that to have the same effective number of digits β on two different problem sizes, say n_1 and n_2 with $n_1 < n_2$, the range R should be different. On these two problems, R needs to be n_1^β and n_2^β, respectively, giving $n_1^\beta < n_2^\beta$.

We need to point out that the integer range R can also be represented by a number in other bases, such as binary. Which base to use will not affect the results quantitatively, but introduces a constant factor to the results. In fact, since $b = \log_{10}(R)$, where R is the range of integers to be chosen, $\beta(n, b) = b/\log_{10}(n) = \log_n(R)$, which is independent of the base of the values for intercity distances.

It is interesting to note that, controlled by the effective number of digits $b/\log_{10}(n)$, the fraction of distinct entities ρ has a property similar to a phase transition, also shown in Figure 35(a). The larger the problem, the sharper the transition, and there exists a crossover point among the transitions of problems with different sizes. We may examine this phase-transition phenomenon more closely using finite-size scaling. Finite-size scaling [5, 101] is a method that has been successfully applied to phase transitions in similar systems of different sizes. Based on finite-size scaling, around a critical parameter (temperature) T_c, problems of different sizes tend to be indistinguishable except for a change of scale given by a power

law in a characteristic length, which is typically in the form of $(T - T_c)n^{1/v}$, where n is the problem size and v the exponent of the rescaling factor. Therefore, finite-size scaling characterizes a phase transition precisely around the critical point T_c of the control parameter as the problem scales to infinity. However, our analysis revealed that the scaling factor has a large exponent of 9 [110], indicating that the phase transitions in Figure 35(a) does not exactly follow the power law finite-size scaling.

To find the correct rescaled control parameter, we reconsider (6). As $n \to \infty$ and distance range R grows with problem size n, i.e., $R \to \infty$ as $n \to \infty$, we can rewrite (6) as

$$\begin{aligned} \lim_{n\to\infty, R\to\infty} E[\rho(n,b)] &= \lim_{R\to\infty} \frac{R}{M}\left(1 - \left((1-1/R)^R\right)^{M/R}\right) \\ &= \frac{R}{M}\left(1 - e^{-M/R}\right), \end{aligned} \tag{7}$$

where the second equation follows $\lim_{R\to\infty}(1-1/R)^R = e^{-1}$. Since our underlying control parameter is the number of digits, $b = \log_{10}(R)$, we take $x = \log_{10}(R/M)$. Asymptotically as $n \to \infty$, $M \simeq n^2$, which leads to $x = \log_{10}(R) - 2\log_{10}(n) = (\beta - 2)\log_{10}(n)$. Using x, we rewrite (7) as

$$\lim_{n\to\infty, R\to\infty} E[\rho(n,b)] = 10^x\left(1 - e^{-10^{-x}}\right). \tag{8}$$

The rescaled control parameter as $n \to \infty$ for the expected number of distinct distances in D is $(\beta - 2)\log_{10}(n)$. Therefore, the critical point is 2 and the rescaling factor is $\log_{10}(n)$. The rescaled phase transition is shown in Figure 35(b), which plots $\rho(n, (\beta - 2)\log_{10}(n))$.

Note that the number of digits used for intercity distances is nothing but a measurement of the precision of the distances. The larger the number of digits, the higher the precision becomes. This agrees with the common practice of using more effective digits to gain precision. Therefore, the control parameter is in turn determined by the precision of intercity distances.

Finally, it is important to note that even though the discussion in this section focused on asymmetric cost matrices and the ATSP, the arguments apply to symmetric distance matrices and the symmetric TSP as well. That is, with M revised to $(n^2 - n)/2$, asymptotically as R and n goes to infinity, $\log_{10}(M) \simeq 2\log_{10}(n)$, so that $(\beta - 2)\log_{10}(n)$ is also a rescaled control parameter for the number of distinct distances in symmetric cost matrices.

4.3 Phase Transitions

With the control parameter, the effective number of digits $\beta(n, b)$ for intercity distances, identified, we are now in a position to investigate possible phase transitions in the ATSP and the related assignment problem.

To set forth to investigate these phase transitions, we generated and studied uniformly random problem instances with 100-, 200-, 300- upto 1,000-cities and 1,500-cities. Although we have results from 100-, 200-, 300-, up to 1,000-city as well as 1,500-city problems, to make the result figures readable, we only use the data from 100-, 500-, 1,000- and 1,500-city problems to report the results. For the problem instances considered, intercity distances are independently and uniformly chosen from $\{0, 1, 2, \cdots, R-1\}$ for a given range R, which is controlled by the number of digits b. We varied b from 1.0 to 6.0 for instances

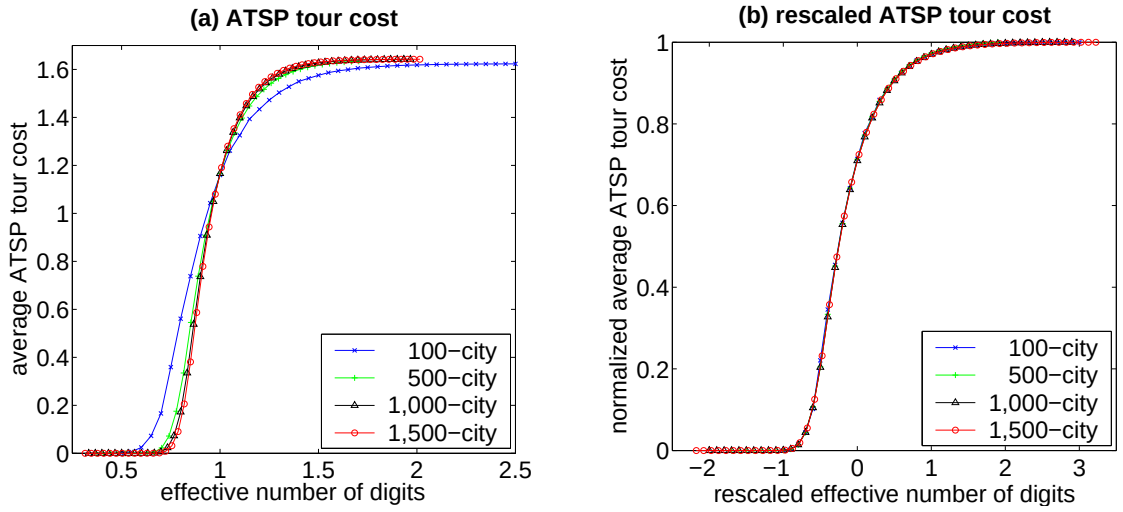


Figure 36: (a) Average optimal ATSP tour cost. (b) Scaled and normalized average optimal tour cost, with rescaling factor $(\beta - \beta_c)\log_{10}(n)$ and $\beta_c = 1$.

with up to 1,000-cities and from 1.0 to 6.5 for instances with 1,500-cities. The digits are incremented by 0.1, i.e., we used $b = 1.0, 1.1, 1.2, \cdots$.

4.3.1 Phase transitions in the ATSP

We are particularly interested in possible phase transitions in the ATSP cost, phase transitions of backbones and phase transitions of the numbers of ATSP tours. The results on backbone can shed some light on the intrinsic tightness of the constraints among the cities as the precision of distance measurement changes.

The ATSP tour cost

There is a phase transition in the ATSP tour cost, $ATSP(D)$, under the control parameter β, the effective number of digits for intercity distances. Figure 36(a) shows the results on 100-, 500-, 1,000- and 1,500-city ATSP instances, averaged over 10,000 instances for each data point. The reported tour costs are obtained by dividing the integer ATSP tour costs by $n \times (R-1)$, where n is the number of cities and R the range of intercity costs. Equivalently, an intercity distance was virtually converted to a real value in $[0, 1]$.

As shown, the ATSP tour cost increases abruptly and dramatically as the effective number of digits increases, exhibiting phase transitions. The transitions become sharper as the problem becomes larger, and there exist crossover points among curves from different problem sizes. By finite-size scaling, we further determine the critical value of the control parameter at which the phase transitions occur. Following the discussion in Section 4.2, the scaling factor has a form of $(\beta-\beta_c)\log_{10}(n)$. Our numerical result indicated that $\beta_c = 1.02 \pm 0.007$. We thus use $\beta_c = 1$ to show the result in Figure 36(b). It is worthwhile to mention that the AP cost follows almost the same phase-transition pattern of the control parameter shown in Figure 35 with a slightly different critical point (data not shown here).

Backbone and number of optimal solutions

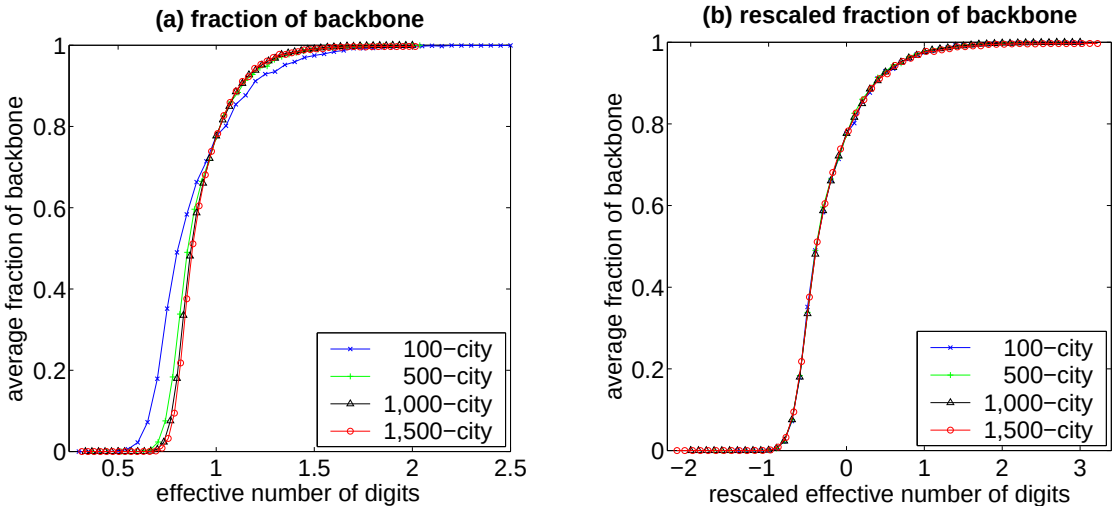


Figure 37: (a) Average fraction of backbone. (b) Rescaled average backbone fraction, with rescaling factor $(\beta - \beta_c)\log_{10}(n)$ and $\beta_c = 1$.

We now turn to the backbone of the ATSP, which is the percentage of directed arcs that appear in all optimal solutions. The backbone also exhibits phase transitions as the effective number of digits for distances increases. The result is included in Figure 37(a), where each data point is averaged over 10,000 problem instances. The rescaled result is shown in Figure 37(b), where the critical point $\beta_c = 1$. Interestingly, the phase-transition pattern of the backbone is almost identical to that of the fraction of distinct entities in the distance matrix, shown in Figure 35, and that of the ATSP tour cost, shown in Figure 36.

The fraction of backbone is related to the number of optimal solutions of a problem. We thus examined the total number of optimal solutions of the ATSP. This was done on small ATSPs, from 10 cities to 150 cities, as finding *all* optimal solutions on larger problems is computationally too expensive. The results are averaged over 100 trials for each data point. As shown in Figure 38, where the vertical axes are in logarithmic scale, the number of optimal tours also undergoes a phase transition, from exponential to a constant, as the number of digits increases. Note that when the number of digits is small, it is very costly to find all optimal solutions, even on these relatively small problems.

The fraction of backbone captures in essence the tightness of the constraints among the cities. As more intercity distances become distinct, the number of tours of distinct lengths increases. Consequently, the number of optimal solutions decreases and the fraction of backbone grows inversely. When more arcs are part of the backbone, optimal solutions become more restricted. As a result, the number of optimal solutions decreases. As the fraction of backbone increases and approaches one, the number of optimal solutions decreases and becomes one as well, which typically makes the problem of finding an optimal solution more difficult.

4.3.2 Existence of Hamiltonian circuit with zero-cost edges

When the precision of intercity distances is low, it is likely that the ATSP tour has a cost of zero, meaning that there exists an Hamiltonian circuit consisting of zero-cost arcs. It is a decision problem to

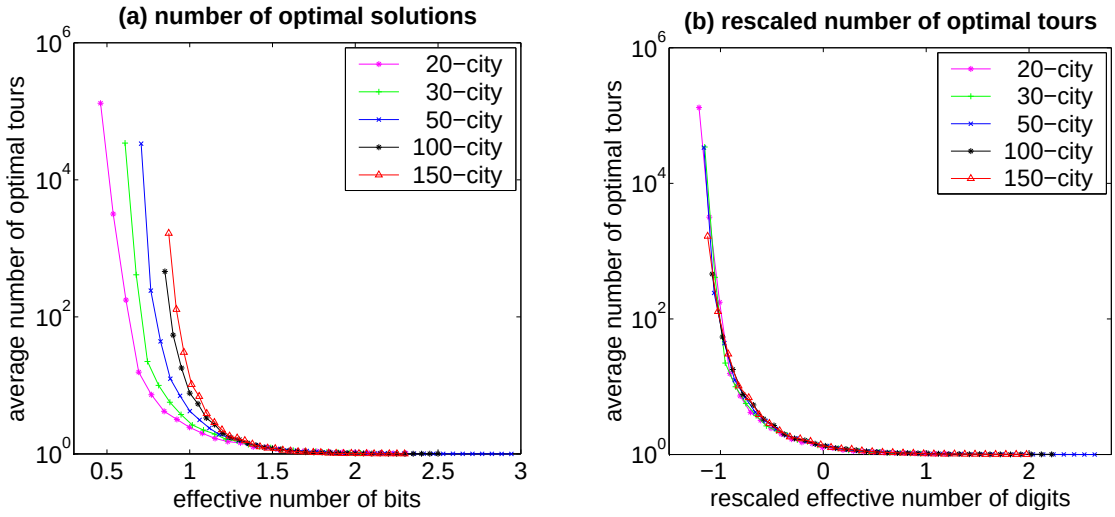


Figure 38: (a) Average number of optimal ATSP tours. (b) Rescaled average number of optimal ATSP tours, with rescaling factor $(\beta - \beta_c)\log_{10}(n)$ and $\beta_c = 1.39 \pm 0.008$.

determine if an Hamiltonian circuit exists in a given ATSP. We examined this decision problem using the same set of 10,000 problem instances used in Figures 36 and 37. The result is shown in Figure 39. Notice that although it follows the same rescaling formula of $(\beta - \beta_c)\log_{10}(n)$, the critical point of the transition, $\beta_0 = 0.865$, is different from the critical point of $\beta_c = 1$ for the phase transitions of backbones and ATSP tour cost, as shown in Figures 36 and 37.

4.3.3 Quality of the AP lower-bound function

The existence of Hamiltonian circuits of zero-cost arcs also indicates that when the number of digits for intercity distances is very small, for example, less than 1.9 (or $R \approx 80$) for $n = 1,500$, both the AP and ATSP costs are zero, meaning that these two costs are the same as well. It is useful to know how likely the AP cost is equal to the ATSP tour cost; the answers to this issue constitutes the first step to the elucidation of the accuracy of the AP lower-bound cost function.

Given a random distance matrix D, how likely is it that an AP cost will be the same as the ATSP tour cost as the effective number of digits β increases? We answer this question by examining the probability that an AP cost $AP(D)$ is equal to the corresponding ATSP cost $ATSP(D)$ as β increases. Figure 40(a) shows the results on 100-, 500-, 1,000- and 1,500-city ATSP instances, averaged over the same set of 10,000 instances for Figure 36 for each data point. As shown in the figure, the probability that $AP(D) = ATSP(D)$ also experiences abrupt and dramatic phase transitions. Figure 40(b) shows the phase transitions after finite-size scaling, with critical point $\beta_c = 1.17 \pm 0.005$.

The results in Figure 40 also imply that the quality of the AP lower-bound function degrades as the distance precision increases. The degradation should also follow a phase-transition process. This is verified by Figure 41, using the data from the same set of problem instances. Note that the critical point of the phase transition for the accuracy of AP is $\beta_c = 0.97$, which is different from the critical point $\beta_c = 1.17$

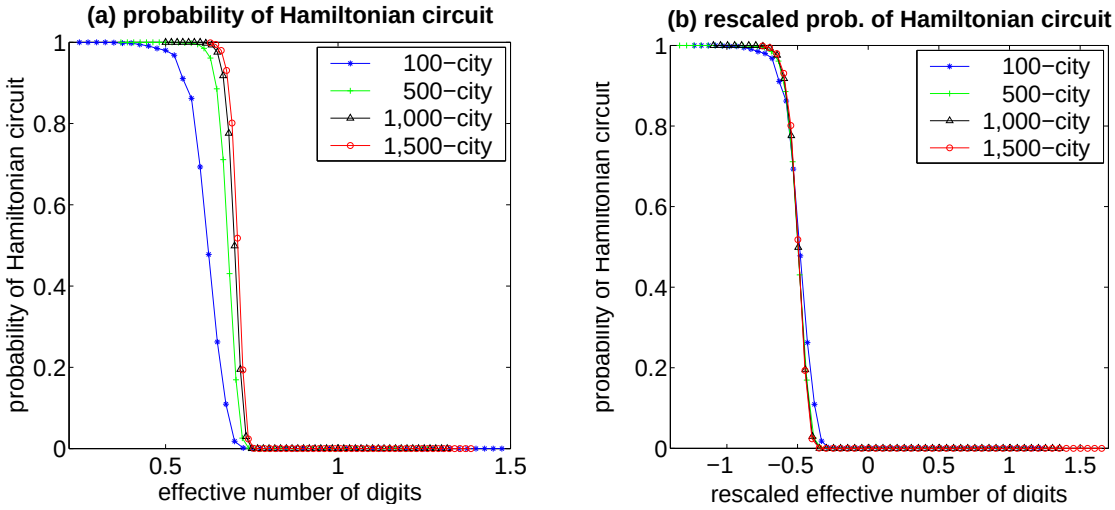


Figure 39: (a) Probability of the existence of Hamiltonian circuits with zero cost arcs. (b) Rescaled probability of zero-cost Hamiltonian circuits, with rescaling factor $(\beta - \beta_c)\log_{10}(n)$ and $\beta_c = 0.865$.

for the phase transition of the probability that $AP(D) = ATSP(D)$.

4.3.4 How many phase transitions?

So far, we have seen many phase transitions on different features of the ATSP and its related assignment problem. Qualitatively, all these phase transitions follow the same transition pattern, meaning that they can all be captured by the same finite-size rescaling formula of $(\beta - \beta_c)\log_{10}(n)$, where β_c is a critical point depending on the particular feature of interest.

It is interesting to note that the critical points for the phase transitions of the ATSP tour costs and backbone fractions are all at $\beta_c = 1$. A close examination also indicates that these two phase transitions follow almost the same phase transition, as shown in Figure 42, where the rescaled curves for the ATSP tour cost and the backbone fraction are drawn from 1,500-city ATSP, averaged over 10,000 problem instances.

Except the close similarity of the phase transitions of the ATSP tour cost and the fraction of backbones, the other phase transitions all have different critical points, indicating that they undergo the same type of phase transitions at different ranges.

4.4 Asymptotic ATSP tour length and AP precision

As a by-product of the phase-transition results, we now provide numerical values of the ATSP cost, the AP cost and its accuracy, asymptotically when the number of cities grows. We attempt to extend the previous theoretical results on the AP cost, which was shown to asymptotically approach $\pi^2/6$ [3, 74], and the observations that the relative error of the AP lower bounds descreases as the problem size increases [4, 92].

Not every real number can be represented in a digital computer. Thus, it is infeasible to directly examine a theoretical result on reals using a digital computer. For our purpose, on the other hand, the phase-transition results indicate that when the precision of the intercity distances is high enough, all the

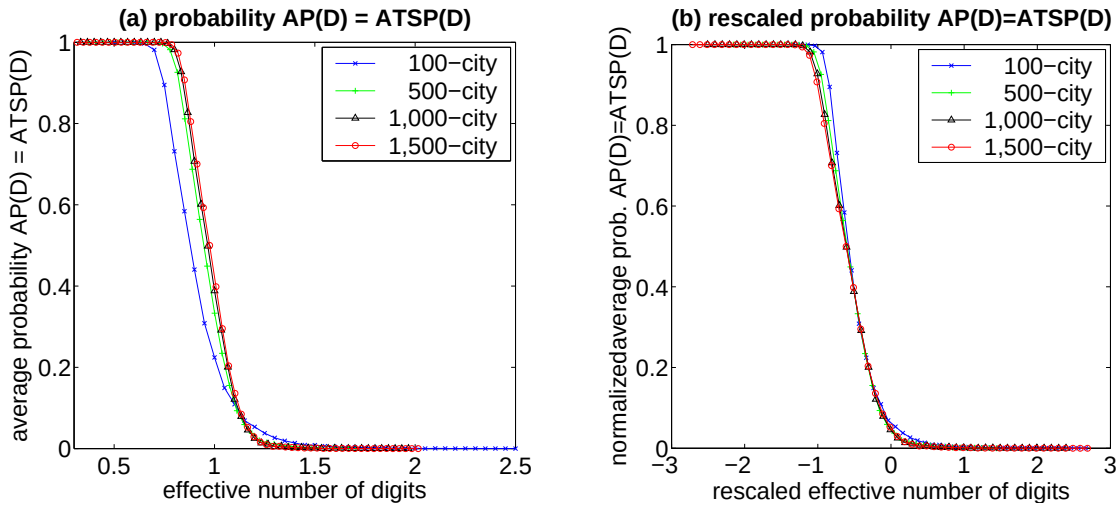


Figure 40: (a) Average probability that $AP(D) = ATSP(D)$. (b) Average probability after finite-size scaling, with rescaling factor $(\beta - \beta_c)\log_{10}(n)$ and $\beta_c = 1.17 \pm 0.005$.

quantities of the ATSP we have examined, including the ATSP cost, the AP cost and its precision as a lower-bound cost function, as well as the backbone, are relatively stable, in the sense that they do not change significantly even when the precision of intercity distances increases further. Therefore, it is sufficient to use a high distance precision to experimentally analyze the asymptotic properties of the ATSP cost and other related quantities.

We need to be cautious in selecting the number of digits for intercity distances. As we discussed in Section 4.2, the same number of digits for intercity distances gives rise to different effective numbers of digits on problems of different sizes. Furthermore, the phase transition results in Section 2.3.2 indicate that the effective numbers of digits must be scaled properly in order to have the same effect on problems of different sizes when we investigate an asymptotic feature.

Therefore, in our experiments, we fixed the scaled effective number of digits for intercity distances, $(\beta - \beta_c)\log_{10}(n)$, to a constant. Based on our phase-transition results, especially that on the control parameter in Figure 35, we chose to take $(\beta - 2)\log_{10}(n)$ a constant of 2.1, for two reasons. First, $(\beta - 2)\log_{10}(n) = 2.1$ is sufficiently large so that almost all distances are distinct, regardless of problem size, and the quantities we will examine will not change substantially after the finite-size scaling. Secondly, $(\beta - 2)\log_{10}(n) = 2.1$ is relatively small so that we can experiment on problems of large sizes. To save memory as much as possible, the intercity distances are integers of 4 bytes in our implementation of the subtour elimination algorithm. Thus the number of digits must be less than 9.4 without causing an overflow in the worst case. Using $(\beta - 2)\log_{10}(n) = 2.1$, we can go up to roughly 3,000-city ATSPs.

Table 3 shows the experimental results, with up to 3,000 cities, on the average AP cost, the ATSP tour cost, and accuracy of the AP cost function in the error of AP cost relative to the ATSP cost. The results are averaged over 10,000 instances for each problem size. Based on the results, the AP cost approaches to 1.6442 and the ATSP cost to 1.6446. Note that the experimental AP cost of 1.6442 is very close to the theoretical asymptotic AP cost of $\pi^2/6 \approx 1.6449$ [3, 74]. In addition, the accuracy of AP function indeed

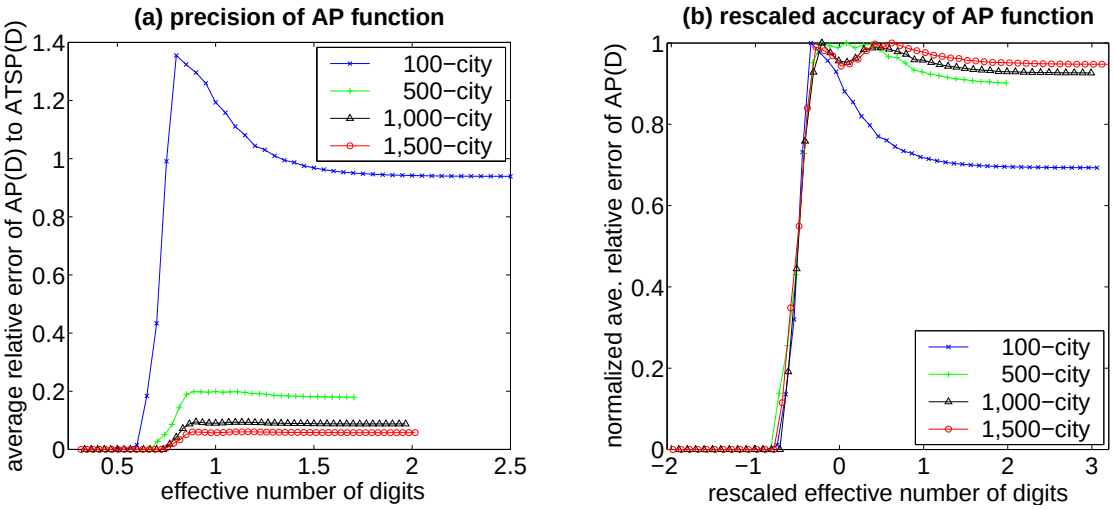


Figure 41: (a) Average accuracy of AP lower-bound function, measured by the error of AP cost relative to ATSP cost. (b) normalized and rescaled average accuracy, with rescaling factor $(\beta - \beta_c)\log_{10}(n)$ and $\beta_c = 0.97$.

improves as the problem size increases, reduced to about 0.02548% for 3,000-city problem instances. This result supports the previous observations [4, 92].

4.5 Threshold Behavior of Subtour Elimination

All the phase-transition results discussed in the previous section indicate that the ATSP becomes more constrained and difficult as the distance precision becomes higher. In this section, we study how a well-known algorithm for the ATSP, branch-and-bound subtour elimination [4, 7, 92], behaves. We separate this issue from the phase transition phenomena studied before because what we will consider in this section is the behavior of a particular algorithm, which may not be necessarily a feature of the underlying problem. Nevertheless, this is still an issue of its own interest because this algorithm is the oldest and is still among the best known methods for the ATSP, and we hope that a better understanding of an efficient algorithm for the ATSP can shed light on the typical case computational complexity of the problem.

4.5.1 Branch-and-bound subtour elimination

The branch-and-bound (BnB) subtour algorithm elimination [4, 7, 92] solves an ATSP in a state-space search [82, 107] and uses the assignment problem (AP) as a lower-bound cost function. The BnB search takes the original ATSP as the root of the state space and repeats the following two steps. First, it solves the AP of the current problem. If the AP solution is not a complete tour, it decomposes it into subproblems by subtour elimination that breaks a subtour by excluding some arcs from a selected subtour. As a subproblem is more constrained than its parent problem, the AP cost to the subproblem must be as much as that of the parent. This means that the AP cost function is monotonically nondecreasing. While solving the AP requires $O(n^3)$ computation in general, the AP to a child node can be incrementally solved in $O(n^2)$ time based on the solution to the AP of its parent.

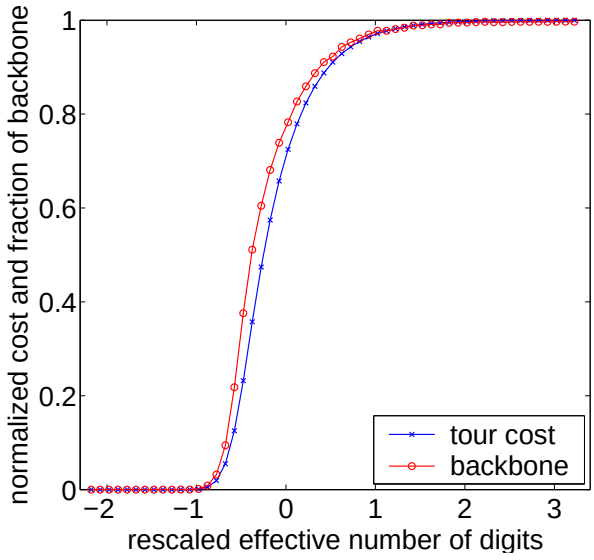


Figure 42: Simultanous examination of the phase transitions of backbone and ATSP tour cost on 1,500-city problems, all rescaled with $(\beta - 1)\log_{10}(n)$.

There are many heuristics for selecting a subtour to eliminate [4], and we use the Carpaneto-Toth scheme [13], or the CT scheme for short, in our algorithm. One important feature of the CT scheme is that it generates no duplicate subproblem so that the overall search space is a tree. One example of this scheme is shown in Figure 43. The AP solution to the original ATSP contains two subtours that are in the root of the tree of the figure. The subtour $2 - 3 - 2$ is chosen to be eliminated, since it is shorter than the other subtour. We have two ways to break the selected subtour, i.e., excluding directed arc $(2, 3)$ or $(3, 2)$. Assume that we first exclude $(2, 3)$ and then $(3, 2)$, generating two subproblems, nodes A and B in Figure 43. When generating the second subproblem B, we deliberately include $(2, 3)$ in its solution. By including the arc that was excluded in the previous subproblem A, we force to exclude in the current subproblem B all the solutions to the original problem that will appear in A, and therefore form a partition of the solution space using A and B. In general, let $\mathcal{E}$ be the excluded arc set, and $\mathcal{I}$ be the included arc set of the problem to be decomposed. Assume that there are t arcs of the selected subtour, $\{e_1, e_2, \cdots, e_t\}$, that are not in $\mathcal{I}$. The CT scheme decomposes the problem into t child subproblems, with the k-th one having excluded arc set $\mathcal{E}_k$ and included arc set $\mathcal{I}_k$, such that

$$\left.\begin{array}{ll} \mathcal{E}_k & = \mathcal{E} \cup \{e_k\}, \\ \mathcal{I}_k & = \mathcal{I} \cup \{e_1, \cdots, e_{k-1}\}, \end{array}\right\} k = 1, 2, \cdots, t. \tag{9}$$

Since e_k is an excluded arc of the k-th subproblem, $e_k \in \mathcal{E}_k$, and it is an included arc of the $k + 1$-st subproblem, $e_k \in \mathcal{I}_{k+1}$, a complete tour obtained from the k-th subproblem does not contain arc e_k, while a tour obtained from the $k + 1$-st subproblem must have arc e_k. Thus a tour from the k-th subproblem cannot be generated from the $k + 1$-st subproblem, and vice versa. In summary, the state space of the ATSP under BnB using the CT subtour elimination scheme can be represented by a tree without duplicate nodes.

In the next step, the algorithm selects as the current problem a new subproblem from the set of *active*

n	digits	AP cost	ATSP cost	relative AP error (%)
200	6.7021	1.63533 ± 0.00254	1.64302 ± 0.00254	0.46817 ± 0.00970
400	7.3041	1.63942 ± 0.00180	1.64311 ± 0.00180	0.22485 ± 0.00468
600	7.6563	1.64072 ± 0.00146	1.64314 ± 0.00145	0.14765 ± 0.00317
800	7.9062	1.64227 ± 0.00125	1.64407 ± 0.00125	0.10904 ± 0.00237
1,000	8.1000	1.64297 ± 0.00114	1.64441 ± 0.00114	0.08754 ± 0.00191
1,200	8.2584	1.64284 ± 0.00104	1.64402 ± 0.00105	0.07187 ± 0.00158
1,400	8.3923	1.64313 ± 0.00096	1.64413 ± 0.00096	0.06148 ± 0.00139
1,600	8.5082	1.64319 ± 0.00090	1.64405 ± 0.00090	0.05276 ± 0.00117
2,000	8.7021	1.64382 ± 0.00082	1.64451 ± 0.00082	0.04231 ± 0.00095
2,200	8.7848	1.64372 ± 0.00077	1.64434 ± 0.00077	0.03813 ± 0.00085
2,400	8.8604	1.64360 ± 0.00074	1.64417 ± 0.00073	0.03477 ± 0.00079
2,600	8.9299	1.64429 ± 0.00071	1.64481 ± 0.00071	0.03234 ± 0.00074
2,800	8.9943	1.64382 ± 0.00068	1.64430 ± 0.00068	0.02966 ± 0.00068
3,000	9.0542	1.64421 ± 0.00065	1.64463 ± 0.00065	0.02548 ± 0.00061

Table 3: Numerical results on AP cost, the ATSP cost and AP error relative to the ATSP cost, in percent. The cost matrices are uniformly random. Each data point is averaged over 10,000 problem instances. In the table, n is the number of cities, digits is the number of digits for intercity distances, and all numerical error bounds represent 95 percent confidence intervals.

subproblems, which are the ones that have been generated but not yet expanded. This process continues until there is no unexpanded problem, or all unexpanded problems have costs greater than or equal to the cost of the best complete tour found so far.

Thanks to its linear-space requirement, we use depth-first branch-and-bound (DFBnB) in our algorithm. DFBnB explores active subproblems in a depth-first order. It uses an upper bound α on the optimal cost, whose initial value can be infinity or the cost of an approximate solution, such as one obtained by Karp's patching algorithm [53, 56], which repeatedly patches two smallest subtours into a big one until a complete tour forms. Starting at the root node, DFBnB selects a recently generated node x to examine next. If the AP solution of x is a complete tour, then x is a leaf node of the search tree. If the cost of a leaf node is less than the current upper bound α, α is revised to the cost of x. If x's AP solution is not a complete tour and its cost is greater than or equal to α, x is pruned, because node costs are monotonic so that no descendant of x will have a cost smaller than x's cost. Otherwise, x is expanded, generating all its child nodes. To find an optimal goal node quickly, the children of x should be searched in an increasing order of their costs. In other words we use node ordering to reduce the number of nodes explored. To speed up the process of reaching a better, possibly optimal, solution, we also apply Karp's patching algorithm to the best child node of the current node.

Our algorithm is in principle the same as that of [12], which is probably the best known complete algorithm for the ATSP. The main difference between the two is that, due to a consideration on space requirement, we use depth-first search while [12] used best-first search.

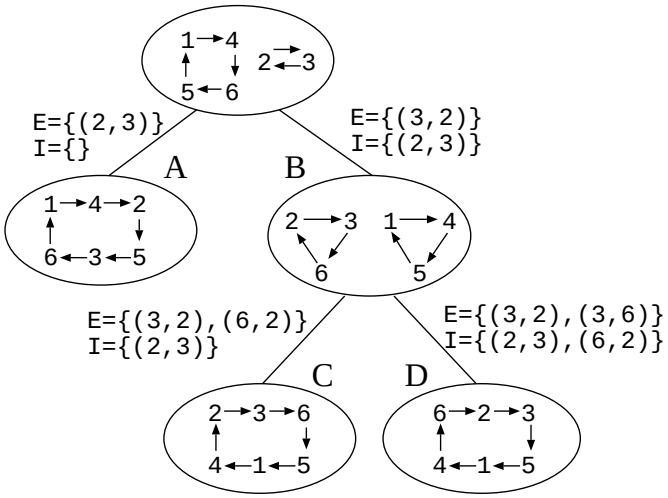


Figure 43: DFBnB subtour elimination on the ATSP.

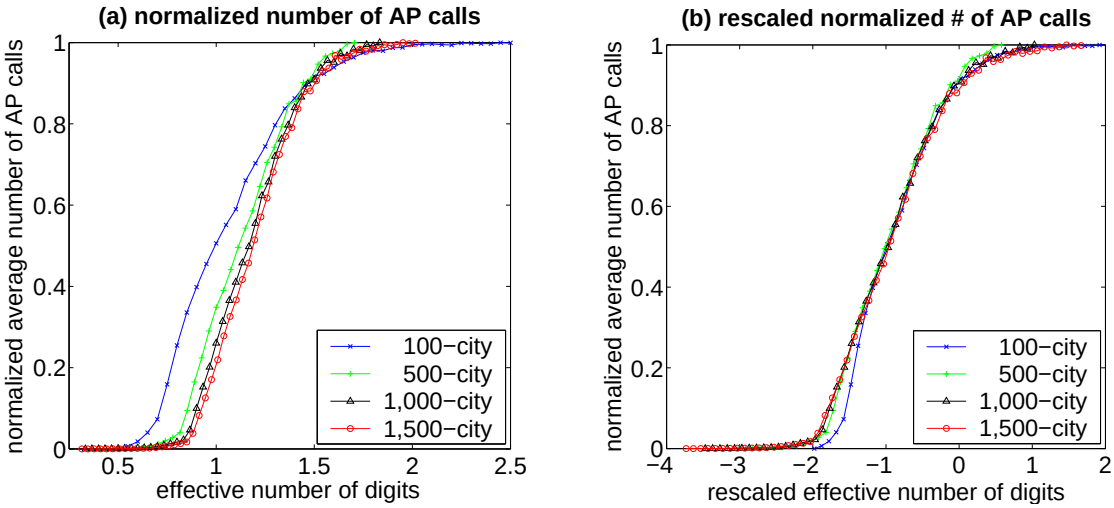


Figure 44: (a) Normalized average number of AP calls of DFBnB subtour elimination. (b) Scaled average number of AP calls, with $(\beta - beta_c)\log_{10}(n)$, where $\beta_0 = 1.49 \pm 0.025$.

4.5.2 Threshing behavior

Figure 44(a) shows the average complexity of DFBnB subtour elimination, measured by the number of calls to the AP function, in terms of the effective number of digits for intercity distances. The result is averaged over the same 10,000 problem instances for each data point as used for the phase transitions studied in Section 2.3.2. Note that the number of AP calls increases significantly from small problems to large ones using the same number of effective digits for distances. Thus, we normalize the result in such a way that for a given problem size, the minimal and maximal AP calls among all problem instances of the same size are mapped to zero and one, respectively, and the other AP calls are proportionally adjusted to a ratio between 0 and 1. This allows us to compare the results from different problem sizes in one figure. The curves in Figure 44(a) follow a pattern similar to that of the phase transitions in Section 2.3.2.

The complexity of the subtour elimination algorithm increases with the effective number of digits, and exhibits a threshold behavior similar to phase transitions. Indeed, we can use finite-size scaling to capture the behavior as the problem size grows, as illustrated in Figure 44(b).

The results in Figure 44 and the phase-transition results of Section 2.3.2 indicate that the complexity of the subtour elimination algorithm goes hand-in-hand with the constrainedness of the problem, which is determined by the portion of distinct entities of distance matrix, which is in turn controlled by the precision of distances.

Similar results have been reported in [113], where the effects of two different distance distributions on the average complexity of the subtour elimination algorithm were analyzed to conclued that the determinant of the average complexity is the number of distinct distances of a problem. The results of this section extend that in [113] to different sizes of problems and by applying finite-size scaling to capture the threshold behavior as problem size increases.

We need to contrast the experimental result in this section with the theoretical result on the NP-completeness of the TSP of intercity distances 0 and 1. It has been known that the degenerated TSP with distances 0 and 1 is still NP-complete [80]. On the other hand, our experimental results showed that when intercity distances are small, relative to the problem size, the ATSP is easy on average. Based on our experimental result, a large portion of the problem instances with small intercity distances can be solved by the assignment problem or Karp's patching algorithm with no branch-and-bound search required. This discrepancy indicates that the worst case of the problem is rare and most likely pathological.

4.6 Related Work and Discussions

Two lines of previous work have directly influenced and inspired this research. The first line of related work was on the expected complexity of tree search, which shed light to the BnB subtour elimination algorithm described in Section 4.5.1 as it solves the ATSP in a tree search. The analysis was carried out on an abstract random tree model called *incremental tree* T [55, 72, 73, 112, 107]. The internal nodes of T has variable number of children and edges in T are assigned finite and nonnegative random values. The cost of a node in T is the sum of the edge costs along the path from the root to that node. An optimal goal node is a node of minimum cost at a fixed depth d. The overall goal is to find an optimal goal node.

There exist phase transitions in the cost of the optimal goal node and the complexity to the problem of finding an optimal goal in T. The control parameter is the expected number of child nodes of a common parent node which have the same cost as the parent. The cost of an optimal goal node almost surely undergoes a phase transition from a linear function of depth d to a constant when the expected same-cost children of a node increases beyond one. Meanwhile, best-first search and depth-first branch-and-bound also exhibit a phase-transition behavior, i.e., their expected complexity changes dramatically from exponential to polynomial in d as the expected same-cost children of a node is reduced to below one. Note that following the result of [21], best-first search is optimal for searching this random tree among all algorithms using the same cost function, in terms of number of node expansions, up to tie breaking. Thus, the above results also give the expected complexity of the problem of searching an incremental tree.

The second line of related research was on characterizing the the assignment problem (AP) lower-bound cost function and its relationship with the ATSP, which has been a research interest for a long time [3, 18, 30, 31, 54, 56, 74, 96]. The first surprising result [96] is that the expected AP cost approaches a constant as the number of cities n goes to infinity if the entries of distance matrix D are independent and uniform over reals $[0, 1]$. This constant has been the subject of a long history of pursuit. It has been shown rigorously, based on rigorous replica method from statistical physics [75], that the optimal cost of random assignment approaches asymptotically to $\pi^2/6$ [3], which is approximately 1.64493. Our results in Section 4.4 show that the AP and the ATSP costs approach 1.64421 and 1.64463, respectively, which support the theoretical results on the AP cost.

More importantly, the relationship between the AP cost and the ATSP cost has remarkably different characteristics under different distance distributions. On one extreme, the AP cost is the same as the ATSP cost with a high probability, while on the other extreme, it can differ from the ATSP cost, with a high probability, by a function of problem size n. Let $AP(D)$ be the AP cost and $ATSP(D)$ the ATSP cost under a distance matrix D. If the expected number of zeros in a row of D approaches infinity when $n \to \infty$, then $AP(D) = ATSP(D)$ with a probability tending to one [30]. However, if the entities of D are uniform over the integers $[0, 1, \cdots, \lfloor c_n n \rfloor]$, then $AP(D) = ATSP(D)$ with a probability going to zero, where c_n grows to infinity with n [30]. Indeed, when the entities of D are uniform over $[0, 1]$, $E(ATSP(D) - AP(D)) \geq c_0/n$, where c_0 is a positive constant [31].

These previous results indicate that the quality of the AP function varies significantly, depending on the underlying distance distribution. Precisely, the difference between the AP cost and the ATSP cost has two phases, controlled by the number of zero distances in the distance matrix D. In one phase, the difference is zero with high probability, while in the other phase, the expectation of the difference is a function of the problem size n. Our experimental results in Section 2.3.2 adds to this analysis the existence of a phase transition between these two phases.

The two-phase result on the accuracy of the AP cost function is also in principle consistent with the phase-transition result of incremental random trees. The root of the search tree has a cost equal to the AP cost $AP(D)$ to the problem and an optimal goal node has the ATSP tour cost $ATSP(D)$. If we subtract the AP cost to the root from every node in the ATSP search tree, the root node has cost zero and an optimal goal node has cost equal to $ATSP(D) - AP(D)$. When there are a large number of zero distances in D, there will be a large number of same-cost children, and the AP cost of a child node in a search tree is more likely to be the same as the AP cost of its parent, since AP will tend to use the zero distances. Therefore, it is expected that more nodes in the search tree will have more than one child node having the same cost as their parents.

4.7 Conclusions

The main contributions from our research and the most important lessons we have learned are twofold. First, we answered positively the long-standing question whether the Traveling Salesman Problem (TSP) has phase transitions [61]. We studied this issue on a more general version of the problem, the asymmetric

TSP (ATSP). We empirically showed that many important properties, including the ATSP tour cost and the fraction of backbone, have two characteristically different values, and the transitions between them are rather abrupt and dramatic, displaying a phase-transition phenomenon. The control parameter of the phase transitions is the effective number of digits representing the intercity distances or the precision of distance measure.

Second, our results provide a practical guidance on how to generate difficult random ATSP problem instances and which random instances should be used to compare the asymptotic performance of ATSP algorithms. A current common practice in comparing algorithms when using a random ensemble is to generate problem instances of different sizes with a fixed distance precision. Our phase transition results indicate that the correct way is to use instances of different sizes that have the same or similar features such as the same fraction of backbones. This in turn requires increasing the precision of intercity distances as the problem size grows.

It is important to note that the exact locations of various phase transitions presented here remain to be mathematically determined, using methods probably from statistical physics [70, 75].

5 Configuration Landscape Analysis and Backbone Guided Local Search for Satisfiability and Maximum Satisfiability

Boolean satisfiability or SAT is an archetypical decision problem [32]. Given a set of Boolean variables and a set of clauses, which specify constraints among the variables, the problem is to decide if there exists a variable assignment that satisfies all constraints. Moreover, there are overconstrained problems in the real world in which not all constraints are satisfiable and the objective is to satisfy the maximal number of constraints. Such a problem is called maximum Boolean satisfiability or Max-SAT, which is the optimization counterpart of SAT. Max-SAT is more general and difficult to solve than SAT. The solution to a Max-SAT can be used to answer the question of its decision counterpart. Many real-world problems can be formulated and solved as SAT or Max-SAT, including scheduling, multi-agent cooperation and coordination, and pattern recognition [6, 16, 20, 29].

Recent years have witnessed significant progresses on SAT in two directions. The first is the understanding of problem properties, such as phase transitions [14, 37, 48, 77] and backbones in various difficult combinatorial problems [78, 109], which were defined in Section 4. It has been shown that there exist sharp transitions in the satisfiability of random SAT problem instances as the ratio of the number of clauses to the number of variables (clause/variable or C/V ratio) increases beyond a critical value [14, 77], a phenomenon similar to phase transitions in disordered systems [37, 44, 48]. It has also been observed that the fraction of backbone variables, the ones that have fixed values among all optimal solutions to Max-SAT, increases abruptly as the clause/variable ratio goes across a critical value, yet another phenomenon similar to phase transitions [109].

The second research direction is focused on developing efficient SAT solvers, especially local search algorithms [71, 89, 87]. The best known local search algorithms include WalkSAT [87] and its variations [71]. These algorithms are able to significantly outperform systematic search algorithms on most random problem instances and some problem classes of real applications, solving larger satisfiable problem instances in less time, albeit the former may fail to reach a solution on a particular instance even if such a solution exists. The great success of WalkSAT has subsequently led to the paradigm of formulating and solving difficult problems from other problem domains, such as planning, as satisfiability problems [57]. The SAT-based approach is now among the best methods for planning.

A problem of fundamental interest and practical importance is how to utilize problem structural information, such as that of phase transitions and backbones, in a search algorithm to cope with the high computational cost of difficult problems, as well as to improve the performance of the algorithm. The published work on this topic is limited. [84] developed a transformation method that exploits phase transitions of tree search problems, and [22] proposed a method to incorporate backbone information from SAT in a systematic search algorithm. [95] proposed a heuristic backbone sampling method for generating initial assignments for a local search for Max-SAT. Despite the success of these previous works, much research needs to be carried out on exploiting structural information in order to demonstrate the viability of incorporating such information in search algorithms.

One of the challenges in utilizing the structural information of a problem, such as phase transitions or

backbones, in a search algorithm, is to make the algorithm not only work on random problem instances, but also perform well on *individual problem instances*, especially those from real-world applications. To our knowledge no result on individual real problem instances has been reported in the previous work using structural information. A phase-transition property is a characteristic of a collection of problem instances drawn from a common distribution. Therefore, phase-transition information may only help when solving a set of related problem instances from which structural information can be extracted. For individual problem instances, however, information of an ensemble may be irrelevant. Therefore, new structural information must be acquired and new mechanism of applying such information must be designed.

In this section we are mainly interested in Max-SAT, thanks to its generality and broad applicability in practice, and local search algorithms, the WalkSAT algorithm in particular, which was originally developed for SAT problem instances. We first investigate the configuration landscapes of local minima reached by WalkSAT. Briefly, we define the configuration landscape of a set of local minima as their distribution with respect to their cost and structure differences in reference to all optimal solutions or a particular local minimum. Similar global structures have been analyzed on other optimization problems, such as the Traveling Salesman problem (TSP) and graph bisection problem, and local search algorithms, such as 2-opt for the TSP [9, 61]. Our experimental results on SAT and Max-SAT show that local minima from WalkSAT reside close to one another, forming clusters in configuration landscapes. The results also indicate that WalkSAT is also effective for Max-SAT, finding many high quality local minima that are very close to optimal solutions in terms of cost and structure differences. Although we carry out this analysis in part for the purpose of developing a new method, the work itself and the results are of interest of their own. Our results indicate that WalkSAT is not only an efficient algorithm for SAT, but also a good choice for Max-SAT. To our knowledge, this study appears to provide the first systematic performance analysis on WalkSAT for Max-SAT. Previous research also showed that WalkSAT is an effective method for solving overconstrained Steiner Tree problems [50].

The second main contribution of this section is an innovative and general heuristic method that exploits backbone information to improve the performance of a local search algorithm. The new method is driven by the results on the configuration space analysis of local minima uncovered in the first part of this section. It is also inspired by the previous research on phase transitions [14, 37, 48, 77] and backbones of combinatorial problems [78, 109]. This method is built upon the following working hypothesis: On a problem whose optimal and near optimal solutions form a cluster, if a local search algorithm can reach close vicinities of such solutions, the algorithm will be effective in finding some information of the solution structures, backbone in particular. This implies that the local minima reached by the algorithm must share many parts of the solution structures with the optimal solutions. If we extract such structure information from local minima, we can then use it to adjust the local search in such a way that it moves directly toward the regions of the search space containing high quality solutions. Using Max-SAT and the WalkSAT local search algorithm, we demonstrate how this new method can improve WalkSAT's effectiveness. We empirically show that the new method is effective on random problem instances as well as *real problem instances* from a SAT library (SATLIB [47]), increasing WalkSAT's probability of reaching better solutions.

We proceed as follows in the rest of the section. In Section 5.1, we first describe SAT and Max-SAT, as well as the WalkSAT local search algorithm for both SAT and Max-SAT. In this section, we also describe an extension to WalkSAT that allows a dynamic parameter tuning [46] at runtime, which we utilize to free WalkSAT from reliance on a manually set noise parameter. We investigate the configuration landscapes of local minima from WalkSAT in Section 5.2. We then develop the backbone guided local search algorithm in Section 5.3. We discuss the main idea of this method, consider how it can be incorporated in WalkSAT to make biased moves, and describe ways of capturing backbone information. We then present in Section 5.4 experimental results of backbone guided local search on random problem instances and instances from SATLIB [47]. We conclude in Section 5.5 with some discussions for future work.

An early version of this section appeared in [115]. The software we developed and used in this research is freely available at http://www.cse.wustl.edu/∼zhang/projects/bgwalksat/index.html.

5.1 SAT, Max-SAT, and WalkSAT local search

We provide in this section some background information on SAT and Max-SAT problems and the WalkSAT local search algorithm for SAT [71, 87]. We also review an existing refinement to WalkSAT for SAT [46], which eliminates WalkSAT's dependence on a manually set noise parameter, and demonstrate its efficacy for Max-SAT.

5.1.1 Boolean satisfiability and maximum satisfiability

A *Boolean satisfiability*, or *SAT*, is typically represented by a formula consisting of a set of Boolean variables and a conjunction of a set of disjunctive clauses of literals, which are variables or their negations. A clause is satisfied if one of its literals is set to true, and a formula is satisfied if no clause is violated. A formula defines constraints on the possible combinations of variable assignments in order to satisfy the formula. A SAT problem is to decide if a variable assignment exists that satisfies all the clauses. When not all clauses can be satisfied, the objective is to maximize the number of satisfied clauses and the problem becomes maximum Boolean satisfiability, or Max-SAT, which is an optimization problem. A SAT or Max-SAT with k literals per clause is short handed as k-SAT or Max-k-SAT, respectively. It is known that k-SAT with k being at least three is NP-complete and Max-k-SAT with k at least two is NP-hard [32], meaning that there is no known polynomial algorithm for the problems, and it is unlikely such an algorithm exists.

5.1.2 The WalkSAT local search algorithm

Since the WalkSAT local search algorithm [87] was developed to solve Boolean satisfiability, the existing study of WalkSAT and its variations has mainly concentrated on *satisfiable* SAT instances. As Max-SAT, which includes *satisfiable* as well as *unsatisfiable* instances, is our main focus in this research, we are interested in the effectiveness of WalkSAT in finding optimal solutions to both satisfiable and unsatisfiable instances.

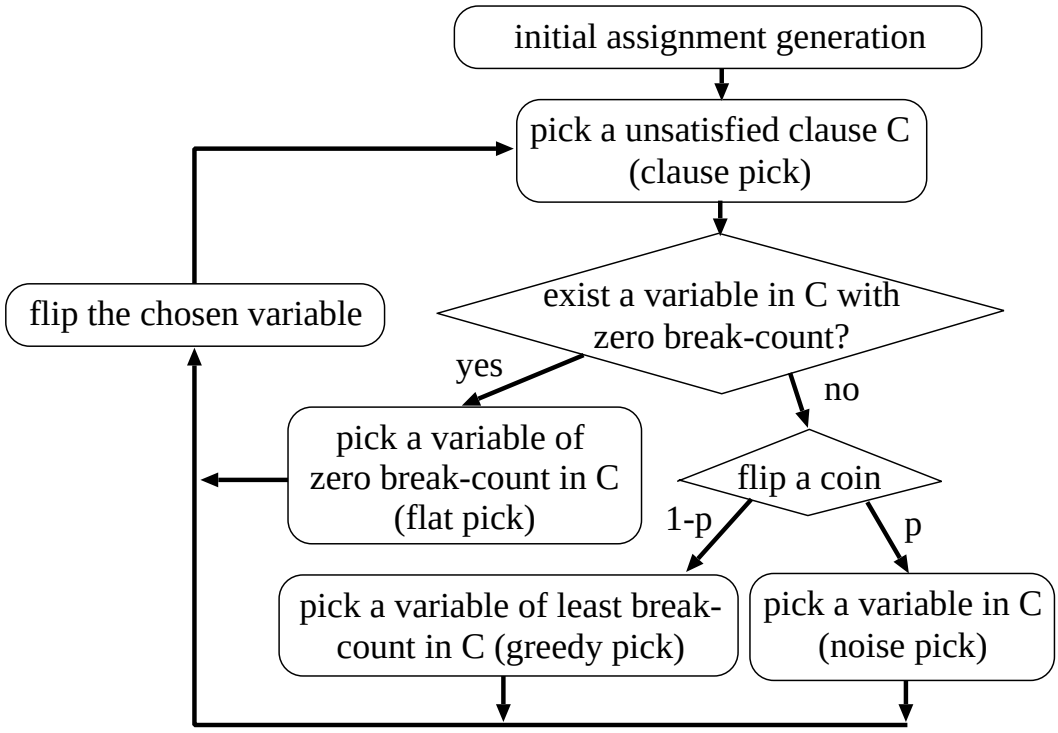


Figure 45: Main operations in a try of WalkSAT.

WalkSAT is a randomized algorithm. The algorithm and its variations all follow the same overall procedure that starts with an initial random variable assignment and makes moves by flipping one variable at a time from True to False or vice versa, until it finds a satisfying assignment or reaches a predefined maximal number of flips. Each such attempt is called a *try* or *restart*. The procedure repeats until a maximal number of tries has been attempted.

To select which variable to flip in each step, the effect of flipping a variable is assessed. Flipping a variable may make some unsatisfied clauses satisfied, and some satisfied clauses unsatisfied. The numbers of clauses that will be made unsatisfied by flipping a variable is called the *break-count* of the variable at the current assignment. WalkSAT attempts to flip a variable with zero break-count, trying to make the next assignment no worse than the current one. To find such a variable with zero break-count, WalkSAT first selects an unsatisfied clause C, *uniformly randomly*, from all unsatisfied clauses. This is called **clause pick**. If C has a variable of zero break-count, WalkSAT then picks such a variable, *uniformly randomly*, from the ones that qualify (called **flat pick**). If no zero break-count variable exists in C, WalkSAT then makes a random choice. With probability p it chooses, *uniformly randomly*, a variable from all the variables involved in C (called **noise pick**); or with probability $1 - p$ it selects a variable with the least break-count, breaking a tie *arbitrarily* if multiple choices exist (called **greedy pick**). Figure 1. The algorithm takes three Figure 45. The algorithm takes three parameters to run: the number of tries, the maximal number of flips in each try, and a probability for noise pick, which is commonly referred to as the *noise ratio* of the algorithm.

5.1.3 WalkSAT with dynamic noise strategy

One limitation of the WalkSAT family of algorithms is their dependence on a manually set noise ratio. To be effective, the noise ratio needs to be tuned for each individual problem, especially for those that do not share common features. So far, two methods have been proposed to resolve this issue for SAT. Auto-WalkSAT [81] uses a probing phase to estimate the optimal parameter for the noise ratio. The estimated noise ratio is then adopted throughout the search phase of the algorithm. Similar to the original WalkSAT, Auto-WalkSAT uses a static noise ratio.

Deviating from the static strategy, WalkSAT with dynamic noise [46] adopts the strategy of automatically adjusting noise ratio as the search progresses. In other words, the dynamic strategy uses different noise ratios at different stages of the search. This strategy seems to be more reasonable than the static strategy. It is relatively easier to make great progress at an early stage of a local search than at a later stage, therefore the noise ratio should be adjusted accordingly, depending on where the current search is in the overall search space.

The idea of dynamic noise strategy is simple: start a local search with the noise ratio equal to zero, and examine the number of violations in the current state every θm flips, where m is the number of clauses of a given problem, and θ a constant. If the number of violations has not decreased since the last time we checked (θm flips ago), the search is assumed to have stagnated, and the noise ratio is increased to $wp + (1 - wp)\phi$, where wp is the current noise ratio and ϕ is another constant. Otherwise, the noise ratio is decreased to $wp(1 - 2\phi)$. The discrepancy between the formulas for increasing and decreasing the noise ratio is based on some empirical observations of how WalkSAT behaves when the noise ratio is too high, compared with how it behaves when the parameter is too low [46]. The dynamic strategy was designed and tested with WalkSAT's cutoff parameter set to infinity; i.e., no random restarts. This is the setting we use for WalkSAT with dynamic noise for all of our experiments for SAT and Max-SAT. For convenience, we refer to this strategy as Dyna-WalkSAT in the remaining of the section.

Note that Dyna-WalkSAT uses two parameters, θ and ϕ. The difference of using these two new parameters and using the noise ratio in the original algorithm is that these two parameters do not have to be tuned for every single problem instance; the performance of Dyna-WalkSAT with the same values for θ and ϕ is relatively consistent across different problem instances. Following [46], we have set $\theta = 1/6$ and $\phi = 1/5$ in our experiments for SAT and Max-SAT.

Though Dyna-WalkSAT was originally designed and tested within the contexts of SAT, we have found it to be effective on Max-SAT as well. In our experiments, we used problem instances of 2,000 variables and C/V ratios of 4.3, 6.0 and 8.0 to capture problem instances from different constrainedness regions. We generated 1,000 random problem instances at each of these C/V ratios. The problem instances were random in that a clause was generated by uniformly picking three literals, without replacement, and by discarding duplicate clauses. For WalkSAT, the noise ratios were set from 0.0 to 0.9, with an increment of 0.1, the number of tries per problem instance at 100, and the number of flips per try at 10,000. We also ran Dyna-WalkSAT on each of these problem instances. To make a fair comparison, we let Dyna-WalkSAT execute one million flips total. Dyna-WalkSAT also used the same parameters as used by WalkSAT, except

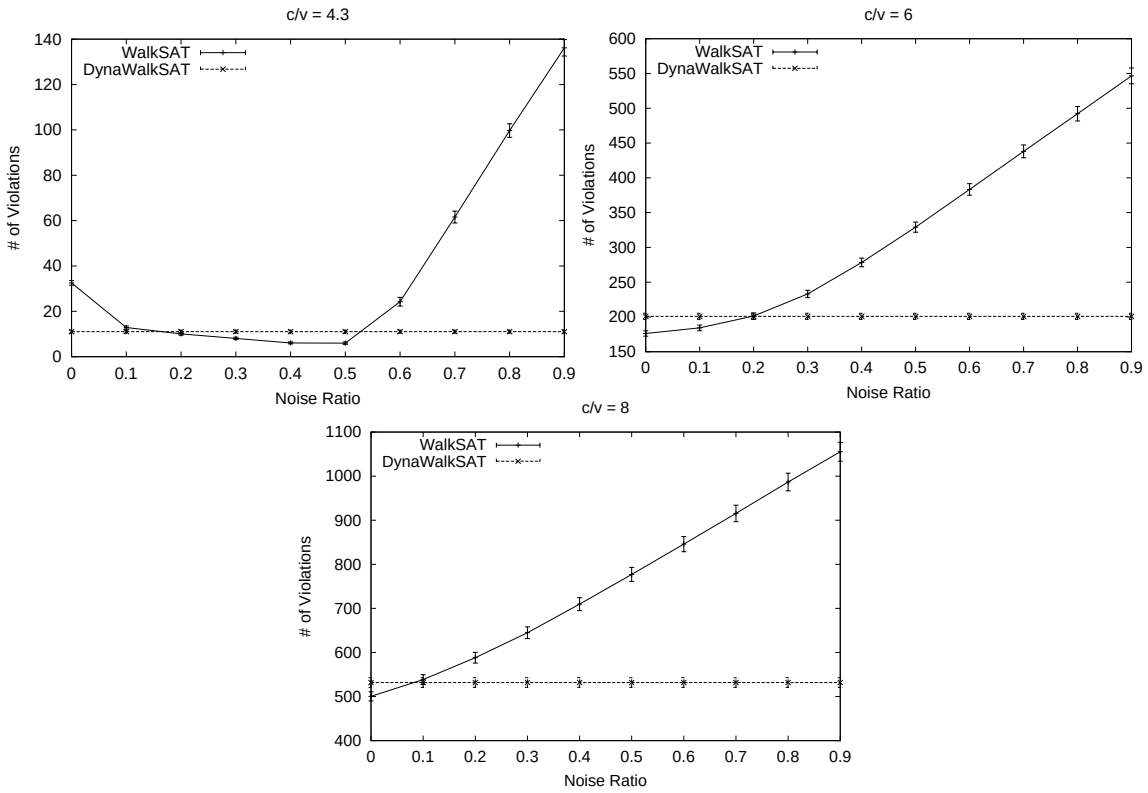


Figure 46: Experimental validation of Dyna-WalkSAT on random Max-3-SAT, for 2,000-variable problem instances.

the noise ratio. To reiterate, following [46] we set $\theta = 1/6$ and $\phi = 1/5$ in Dyna-WalkSAT, which have been found to be effective over a wide range of SAT and Max-SAT instances.

The experimental results are shown in Figure 2. The horizontal axes are noise ratios for WalkSAT and the vertical axes record the average solution quality. The error bars in the figures measure the 95% confidence intervals of the results. For all three C/V ratios tested, the performance of Dyna-WalkSAT is very close to the performance of WalkSAT with the optimal noise ratio, indicating that the dynamic noise strategy is effective for 3-SAT and Max-3-SAT.

Due to its simplicity and reasonable performance, in the rest of this section we will use Dyna-WalkSAT with $\theta = 1/6$ and $\phi = 1/5$ as default parameters to replace WalkSAT in our experimental analysis.

5.2 Configuration Landscapes

Given two variable assignments to a given SAT or Max-SAT problem instance, we can measure their differences in two ways. The first is the difference of their costs or the numbers of violated clauses. This difference can be normalized (divided) by the total number of clauses, giving the difference of violations per clause. The second quantity measures structural difference in the form of the Hamming distance between the two assignments. Since a solution to a SAT problem is simply a string of 0 and 1, Hamming distance here is simply the conventional Hamming distance for binary strings. The Hamming distance can

also be normalized by the total number of variables, resulting in the normalized Hamming distance per variable. We adopt normalized cost difference and normalized Hamming distance to make the results from problems of different sizes directly comparable.

With the relative solution quality and structure difference of two assignments specified, we define the configuration landscape of a set of assignments or solutions as the distribution of the solutions in terms of their qualities and structure differences relative to a reference solution, which can be an optimal solution or the best solution in a given set. The set of solutions can be all the optimal solutions, all suboptimal solutions up to a fixed bound, as well as local minima from a local search.

We can use landscape configuration to capture the effectiveness of the WalkSAT and Dyna-WalkSAT algorithms on 3-SAT and MAX-3-SAT. We carried out two sets of experiments. In the first set of experiments, we aimed to directly measure the effectiveness of WalkSAT in terms of finding optimal and near optimal solutions. To this end, we used all optimal solutions to measure the quality of a set of local minima from WalkSAT. Since finding all optimal solutions is computationally expensive, we restricted ourselves to relatively small random problem instances with 100 variables and C/V ratios of 2.0, 4.3, 6.0 and 8.0 to capture problems in different constrainedness regions. We generated 1,000 problems for each C/V ratio. The problems were randomly generated by uniformly picking three literals without replacement for a clause, with duplicate clauses discarded.

To find all optimal assignments to a Max-SAT problem, we extended the well known Davis-Putnam-Logemann-Loveland (DPLL) algorithm for SAT [19] to Max-SAT. We ran our extended DPLL algorithm for Max-SAT [102] and WalkSAT on the same set of 100-variable Max-3-SAT problem instances. For WalkSAT, we set the number of tries per problem at 100, the number of flips per try at 10,000, and the noise ratio at 0.5. We then examined the configuration landscapes of the local minima reached by WalkSAT against the optimal solutions in terms of the cost difference between a local minimum and an optimal solution as well as the Hamming distance of the local minimum to its nearest optimal solution. Note that the Hamming distance of a local minimum in fact measures the minimal number of flips required to turn the local minimum into an optimal solution.

The configuration landscapes of local minima from WalkSAT are summarized in Figure 47. Since WalkSAT is very efficient on underconstrained SAT instances, finding satisfiable solutions on nearly all problem instances when the C/V ratio is 2.0, we do not include the results for a C/V ratio of 2.0 here. The X-Y planes in the figures show the correlation between the normalized Hamming distance and the normalized cost difference. Each point on the X-Y plane represents a set of possible local minima with the same cost difference and Hamming distance that may be visited by WalkSAT. The origins of the figures correspond to global optima. The vertical Z axes measure the total numbers of local minima reached by WalkSAT.

As shown in Figure 47(a), WalkSAT performs well on underconstrained and critically constrained problems, in that it can find global minima very often. This is shown by the point on the Z axis indicated by the arrow in the figure. Therefore, WalkSAT is effective in finding optimal solutions on underconstrained and critically constrained problems. However, the number of local minima that are also global optima

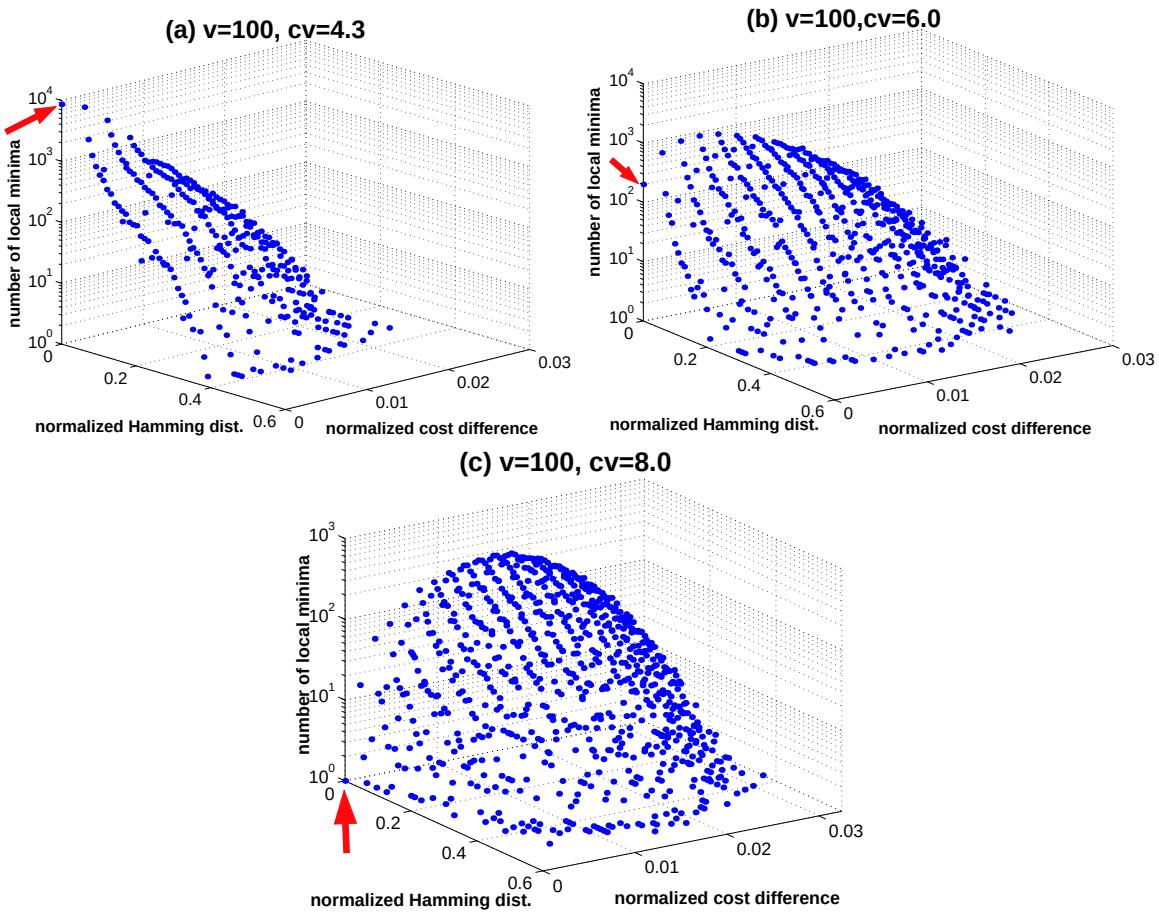


Figure 47: Configuration landscapes of local minima from WalkSAT on 100 variable random 3-SAT and Max-3-SAT, relative to optimal solutions.

decreases from 66,677 to 8,616 as the C/V ratio increases from 2.0 to 4.3, indicating that the effectiveness of WalkSAT decreases. This number decreases further from 201 to 0 on overconstrained problems with C/V ratios of 6.0 and 8.0, respectively (Figures 47(b) and (c)). This result indicates that WalkSAT becomes less effective at finding optimal solutions as problem constrainedness increases. Figure 48 shows the contours of the configuration landscapes in Figures 47(b) and (c) on the X-Y planes, showing a nearly linear correlation between the cost difference and the Hamming distance. That is, a local minimum that has a high cost tends to have a large Hamming distance to an optimal assignment.

In the second set of experiments, we examined the configuration landscapes of local minima from Dyna-WalkSAT. We used 2,000-variable random 3-SAT and Max-3-SAT with C/V ratios of 4.3, 6.0 and 8.0. As before, we generated 1,000 problem instances for each C/V ratio. Because the problems were too large to be solved optimally, we built a configuration landscape of a set of local minima with respect to the best local minimum among them. The results are shown in Figure 49, which are qualitatively similar to the configuration landscapes in Figure 47.

An interesting result from these experiments is that the configuration landscapes of local minima reached by WalkSAT and Dyna-WalkSAT exhibit *bell surfaces* on overconstrained problems with large

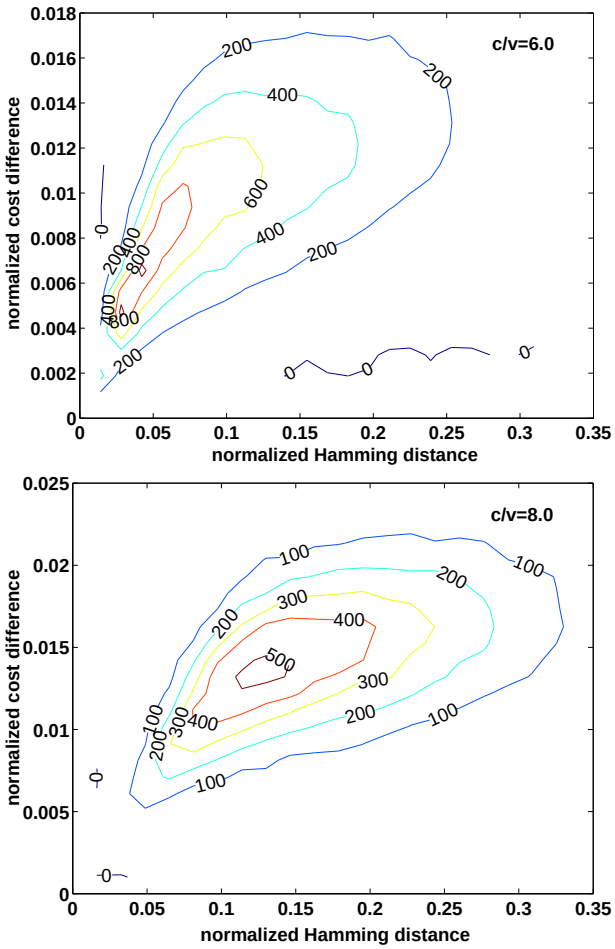


Figure 48: Contours of the configuration landscapes of local minima from WalkSAT on 100 variable Max-SAT with C/V ratios of 6.0 and 8.0.

C/V ratios. More importantly, the summit of such a bell surface shifts away from optimal solutions, the (0,0) point on the X-Y plane, as the C/V ratio increases. This observation is examplified by the contours in Figure 48. Nevertheless, despite the increased difficult of Max-SAT as the C/V ratio grows, WalkSAT and Dyna-WalkSAT are still fairly effective in that they are able to reach local minima that are close to global optima. For 100-variable instances with C/V ratio of 8.0 (Figure 48(b)), the majority of local minima reached by WalkSAT, the ones located at the peak point of the bell surface of the figure, have a normalized cost difference to optimal solutions of 0.014 for 100 variable problems, which is equivalent to about eleven more constraints violated than an optimal solution on such overly constrained problem instances. Since WalkSAT is typically executed with multiple trials, the best local minimum it can landed on will be much better than such most likely local minima.

Another interesting and important observation of the results in Figure 48 is that there is a near linear correlation between the cost difference between a local minimum and its nearest optimal solution and their Hamming distance, the two quality measurements we adopt. This is evident that the contours in Figure 48 have ellipse shapes. An implication of this observation is that a local minimum with a small cost is more

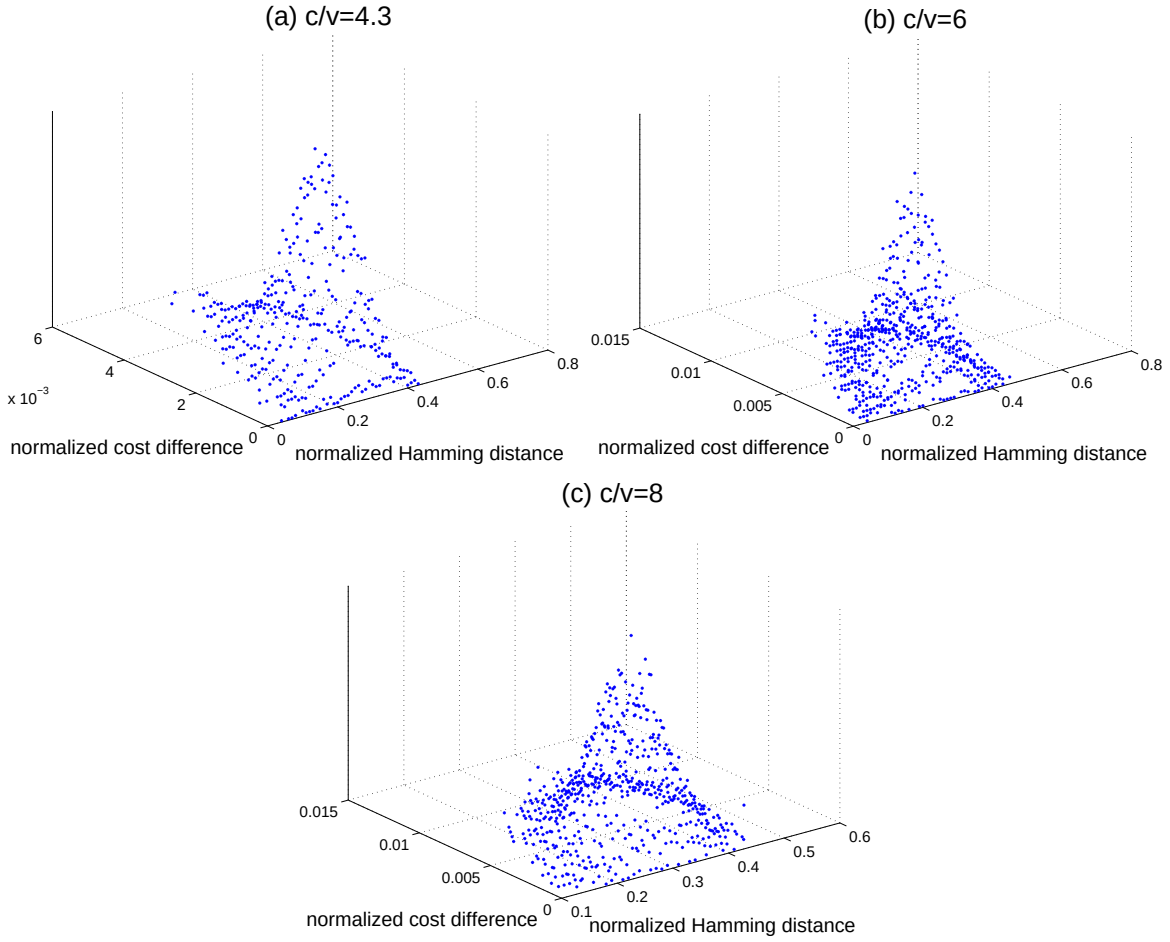


Figure 49: Local minima from WalkSAT on 2,000-variable Max-3-SAT with C/V ratios of 4.3, 6 and 8.

likely to share a larger common solution structure with an optimal solution. More importantly, majority local minima have most parts of their variable assignments consistent with their nearest global optima. For instance, the majority local minima on 100-variable problems with C/V ratio of 8.0 have normalized Hamming distances around 0.13. This means that out of 100 variables, 87 of them are correctly set. This implies that these local minima must share large portions of the variable assignments with the optimal solutions. We will exploit this phenomenon in our new search algorithm in the next section.

In concluding this section, we need to point out that the results of this section extended the previous studies on global structures of optimization cost surfaces [8, 9, 61]. It has been shown in these studies (and other works cited in [8, 9, 61]) that there exists a correlation between the cost differences and distances among local minima of such optimization problems as the symmetric Traveling Salesman problem and graph bisection problems. [8, 9] particularly showed a near linear relation between the quality of local minima and distances among them, which is similar to the contour results in Figure 48. Our results made two noticeable extensions to the study of global structures of optimization problems. First, we considered configuration landscape defined as distribution of local minima with respect to the cost differences and distances between the local minima and optimal solutions or best local minima. This allows us to

directly examine globally convex property or "big valley" structure of the cost surfaces of optimization problems [9]. Secondly, our results showed that there indeed exists a big valley structure in the configuration landscapes of 3-SAT and Max-3-SAT, supporting the "big valley" conjecture on optimization problems [8].

Furthermore, we would also like to point out that the "big valley" results in [8, 9] led to a new multi-start strategy that utilize such a "big valley" structure of local minima. In this section, we present a different, novel way to exploit such global structures, which is the topic of the next section.

5.3 Backbone Guided Local Search

In this section, we discuss in detail the backbone guided local search algorithm. We first present the main idea and then discuss how it can be applied to WalkSAT, forming the backbone guided WalkSAT algorithm. We also consider two different ways to estimate backbone frequencies using local minima.

5.3.1 Main ideas

The backbone variables of a problem are the ones that are critically constrained, since they must be set to particular values to make an optimal solution feasible. By the same token, if a pair of a variable and one of its values appears more often in the set of all optimal solutions, the variable is more constrained. If, somehow, we knew the frequency of a variable-value pair in all optimal solutions, we could construct a "smart" search algorithm by using the backbone frequency information as an oracle to guide each step of the algorithm. Take WalkSAT as an example. At each step of the algorithm, we can use the backbone frequencies to change the way in which a variable is chosen to flip, i.e., we prefer flipping a variable that is unsynchronized with its backbone frequency more than another variable under the current assignment. In other words, we should focus on fixing the critically constrained variables that are not currently set correctly.

Unfortunately, exact backbone frequencies of a problem are even more difficult to come by than actual problem solutions. To address this problem, the second key idea of backbone guided local search is to estimate backbone frequencies using local minima of a local search. We simply treat local minima as if they were optimal solutions and compute *pseudo backbone frequencies*, which are an estimation of real backbone frequencies. More precisely, we define the pseudo backbone frequency of a literal (a variable-value pair) as the frequency with which the literal appears in all local minima, which we denote as $p(l)$ where l is a literal. Note that $p(l) = 1 - p(\neg l)$, where $\neg l$ is the negation of l.

The quality of pseudo backbone frequencies depends on the effectiveness and efficiency of the local search algorithm used. As discussed in Section 5.2, high-quality local minima can be obtained by efficient local search algorithms, such as WalkSAT. Even though WalkSAT may land on suboptimal solution with fairly high probabilities, particularly on overconstrained problem instances, most of the local minima from WalkSAT indeed have large portions of variables set to the correct values, so that they contain parts of optimal solutions or partial backbone. In this research, we adopt WalkSAT to collect local minima and then in return apply the backbone guided search method to WalkSAT to improve its performance.

5.3.2 Biased moves and selections

Pseudo backbone frequencies can be incorporated in a local search algorithm to make "biased" moves or flips. Consider a simple example of two variables, x_1 and x_2, that appear in a violated clause and have the same effect under the current assignment, i.e., flipping one of them makes the violated clause satisfied, and both variables have the same break-count or will cause the same number of satisfied clauses unsatisfied if flipped. Let B be the set of backbone variables along with their fixed values, $\mathcal{T}$ be the set of local minima from which pseudo backbone frequencies were computed, and v_1 and v_2 are the current values of x_1 and x_2. We will prefer to flip x_1 over x_2 if under the current assignment, $P\{(x_1 = v_1) \in B|\mathcal{T}\} < P\{(x_2 = v_2) \in B|\mathcal{T}\}$, which means that under the current assignment, x_1 is less likely to be part of backbone than x_2, given the set of local minima $\mathcal{T}$. Note that $P\{(x = v) \in B|\mathcal{T}\}$ is the pseudo backbone frequency of literal $x = v$ under the evidence of a set of local minima $\mathcal{T}$.

How can the pseudo backbone frequencies be used to alter the way that WalkSAT chooses variables? As discussed in Section 5.1.2, WalkSAT makes *uniformly* random choices in selecting a variable to flip when multiple choices exist. For example, when there are more than one unsatisfied clause under the current assignment, WalkSAT arbitrarily (uniformly) chooses an unsatisfied clause. Similarly, when there are multiple variables with zero break-count, WalkSAT chooses one arbitrarily.

Based on the maximum entropy principle [49], it is optimal on average to make an unbiased choice if there is no information to distinguish one choice over another. Therefore, with no additional information on optimal assignments, the WalkSAT algorithm is optimal on average in terms of selecting a variable to flip. In backbone guided search, we apply pseudo backbone information to force WalkSAT to make random but *biased* choices. If a backbone variable and a nonbackbone variable can make a clause satisfied, the backbone variable should be chosen. In other words, we modify WalkSAT's random strategies in such a way that a backbone or overconstrained variable will be chosen more often than a nonbackbone variable. To this end, we use pseudo backbone frequencies to help make random biased selections.

5.3.3 Backbone guided WalkSAT

Backbone guided WalkSAT has two phases. The first is a *estimation* phase that collects local minima by running WalkSAT, with a fixed number of tries. The local minima thus collected are compiled to compute the pseudo backbone frequencies of all literals.

The second phase carries out the actual *backbone guided* local search, which uses pseudo backbone frequencies to modify the way that WalkSAT chooses variables to flip. The second phase also runs many tries, each of which produces a local minimum, very often a new one. The newly discovered local minima can be subsequently added to the pool of all local minima found so far and be used to update the pseudo backbone frequencies.

We now consider methods for making biased moves in WalkSAT. The first random choice in WalkSAT is clause pick, where an unsatisfied clause is selected when multiple ones exist. We want to pick, with high probabilities, those variables that are either part of the backbone or highly constrained in all optimal solutions. Therefore, we should choose a clause containing the maximal number of critically constrained

variables. To this end, we use the total pseudo backbone frequency of all the literals in an unsatisfied clause, normalized among all unsatisfied clauses, to measure the likelihood that the clause contains backbone and highly constrained variables. We then select an unsatisfied clause among all unsatisfied based on their likelihoods of containing backbone variables. Specifically, let $\mathcal{C}$ be the set of unsatisfied clauses, and q_C be the sum of pseudo backbone frequencies of all the literals in a clause $C \in \mathcal{C}$. We then let $p_C = q_C/Q$ be the probability to select clause C among all unsatisfied clauses in $\mathcal{C}$, where $Q = \sum_{C \in \mathcal{C}} q_C$ is a normalization factor.

WalkSAT uses three other random pick rules to arbitrarily select a variable after an unsatisfied clause is chosen (see Section 5.1.2 and Figure 45). To reiterate, the flat pick rule chooses a variable from a set of zero break-count variables, if any; the noise pick rule selects one from all variables involved in the chosen clause; and the greedy pick rule takes a variable among the ones of least break-count. In essence, these rules use the same operation, picking a variable equally likely from a set of variables. Therefore, we can modify these rules all in the same way by using pseudo backbone frequencies. Let $\{x_1, x_2, \cdots, x_w\}$ be a set of w variables from which one must be chosen, $\{v_1, v_2, \cdots, v_w\}$ their current assignments, and $\{p_1, p_2, \cdots, p_w\}$ the pseudo backbone frequencies of literals $\{(x_1 = v_1), (x_2 = v_2), \cdots, (x_w = v_w)\}$. Then we choose variable x_i with probability $(1 - p_i)/\sum_{j=1}^{w}(1 - p_j)$. Here we use probability $1 - p_i$ because it is the probability of literal $(x_i = \neg v_i)$ being in the pseudo backbone, which is the value x_i is going to change to.

Furthermore, the idea of pseudo backbone frequencies can also be applied to generate an initial assignment for a local search. Specifically, a variable is assigned a particular value with a probability proportional to the pseudo backbone frequency of the variable-value pair. This was called heuristic backbone sampling in [95].

5.3.4 Backbone guided WalkSAT with dynamic noise

To make the backbone guided WalkSAT algorithm more general and robust, we would like to have it use dynamic noise strategy. The dynamic noise strategy discussed in Section 5.1.3 runs a long sequence of variable selections and flips with no restarts. This, unfortunately, is incompatible with backbone guided local search, which requires random restarts in order to collect local minima to construct pseudo backbone frequencies.

To overcome this problem, we have devised a "compromise", which allows a reasonable combination of using dynamic noise method and applying backbone information. Specifically, we run WalkSAT with dynamic noise strategy for a number of short runs to construct pseudo backbone frequencies, followed by several long runs of backbone guided local search. In our particular implementation of backbone guided Dyna-WalkSAT, we let it run thirty short runs for computing pseudo backbone frequencies, followed by seven long runs, each of which has ten times more flips than a short run.

5.3.5 Computing pseudo backbone frequencies

The performance of backbone guided local search depends greatly upon the quality of pseudo backbone information used. The more truthful the pseudo backbone information is, the more effective the new local search will be. In order to retrieve as much backbone information as possible, an unbiased sample of local minima should be used, in which local minima need to be derived from independently generated starting assignments. Therefore, random initial assignments should be preferred.

Given a set of local minima, the computation of pseudo backbone frequencies needs to be done with care. We propose two different ways to compute pseudo backbone frequencies. The first and most straightforward method is to treat all the given local minima as if they were of equal quality, and take the frequency of a literal l that appears in all local minima S as its pseudo backbone frequency $p(l)$. Specifically, we have

$$p(l) = \frac{\sum_{\forall s_i \in S, l \in s_i} 1}{|S|} \tag{10}$$

We call this method *averaging counting* or AC for short.

It is imperative to note that not all local minima are of equal quality. In general, a lower quality local minimum tends to contain less backbone information than a higher quality local minimum, as discussed in Section 5.2. Therefore, the backbone information carried in a lower quality local minimum is less reliable. This means that a literal appearing in a lower quality local minimum should contribute less to the pseudo backbone frequencies than a literal appearing in a higher quality local minimum. As a result, we introduce a discount factor to adjust the contribution of a literal based on the quality of the local minimum where it came from. If a local minimum s_i has cost c_i, which is the number of violated clauses in the local minimum, then we can compute the pseudo backbone frequency $p(l)$ of a literal $l = (x_i = v_i)$ as follows.

$$p(l) = \frac{\sum_{\forall s_i \in S, l \in s_i} \left(\frac{1}{c_i}\right)}{\sum_{\forall s_i \in S} \left(\frac{1}{c_i}\right)} \tag{11}$$

In other words, the contribution of a local minimum toward a pseudo backbone probability of a literal is reciprocally weighted by the cost of the local minimum. We thus call this method *cost reciprocal averaging counting* (CRAC).

5.4 Experimental Evaluation

We now experimentally analyze the performance of the backbone guided WalkSAT (BG-WalkSAT) algorithm on SAT and Max-SAT. We used the dynamic noise strategy in both WalkSAT and BG-WalkSAT, i.e., we compare Dyna-WalkSAT and BG-Dyna-WalkSAT. Our benchmark problems are randomly generated problems and those from the SATLIB [47].

5.4.1 Random ensembles

In this set of experiments, we generated random MAX-3-SAT instances with 2,000 variables and three different C/V ratios of 4.3, 6.0, and 8.0, to sample instances from regions of differing constrainedness. We

Configuration	*C/V ratio*		
	4.3	*6.0*	*8.0*
Dyna-WalkSAT	11.06 ± 0.718	200.87 ± 4.411	531.79 ± 10.990
BG-Dyna-WalkSAT	23.28 ± 1.060	190.91 ± 4.958	504.21 ± 10.643
BG-GreedyPick	20.58 ± 1.046	200.01 ± 4.499	506.68 ± 10.638
BG-NoisePick	10.47 ± 0.725	185.21 ± 4.203	518.48 ± 10.827
BG-Initialization	7.33 ± 0.627	190.290 ± 4.19	519.96 ± 10.771
BG-ClausePick	13.75 ± 0.693	202.74 ± 4.595	537.02 ± 11.156

Table 4: Comparison of backbone guided Dyna-WalkSAT variations over Dyna-WalkSAT on 2,000 variable random Max-3-SAT. Performance is measured by the average number of constraint violations. The errors represent 95% confidence intervals.

ignored C/V ratio of 2.0 since WalkSAT can easily find satisfiable assignments to most of such underconstrained problems. At each of the ratios considered, we generated 1,000 problem instances, and compared Dyna-WalkSAT and BG-Dyna-WalkSAT. Dyna-WalkSAT ran one long try with a maximum of one million flips. BG-Dyna-WalkSAT ran 30 short tries, each with a maximum of 10,000 flips, as in WalkSAT, to collect local minima. It then executed seven long tries, each with 100,000 flips, which was ten times longer than a short try. Thus BG-Dyna-WalkSAT executed one million flips also.

We found that on random problem instances the average counting (AC) method for computing pseudo backbone frequencies is less effective than the cost reciprocal averaging counting (CRAC) method under all different C/V ratios we tested. Therefore, we will present the results from CRAC here.

Using random problem instances, we first examined the effects of applying biased moves to Dyna-WalkSAT. The results are included in Table 4. In the table, we list the results of average constraint violations for Dyna-WalkSAT and BG-Dyna-WalkSAT first, followed by the applications of biased moves to different random picks in WalkSAT. For instance, BG-ClausePick stands for Dyna-WalkSAT with biased clause picks. As shown, biased noise pick and biased initialization can improve Dyna-WalkSAT under all C/V ratios; biased clause pick has negative effects on performance; and biased greedy pick is only effective on highly overconstrained problems. Their combination also has mixed effects: the combined biased moves improve upon Dyna-WalkSAT on overconstrained Max-3-SAT. Note that we do not include the results for biased flat pick because it has no effect on almost all problem instances we tested. The reason is that in most cases, there is only one variable with zero break-count for 3-SAT, so that biased flat pick was not used most of the time.

The results in Table 4 also show that BG-Dyna-WalkSAT can only improve upon Dyna-WalkSAT on overconstrained instances with the C/V ratios equal to 6.0 and 8.0; while it fails to do so on critically constrained instances with the C/V ratio of 4.3. One possible reason is that Dyna-WalkSAT is very effective and efficient, finding optimal solutions very often.

Additional insight can be gained from an inspection of anytime behavior of Dyna-WalkSAT and BG-Dyna-WalkSAT, which are shown in Figure 50. As discussed earlier, BG-Dyna-WalkSAT outperforms Dyna-WalkSAT only when the C/V ratios are 6.0 and 8.0. A key observation on these figures is that there was a big jump on the quality of the best local minimum found so far right after pseudo backbone

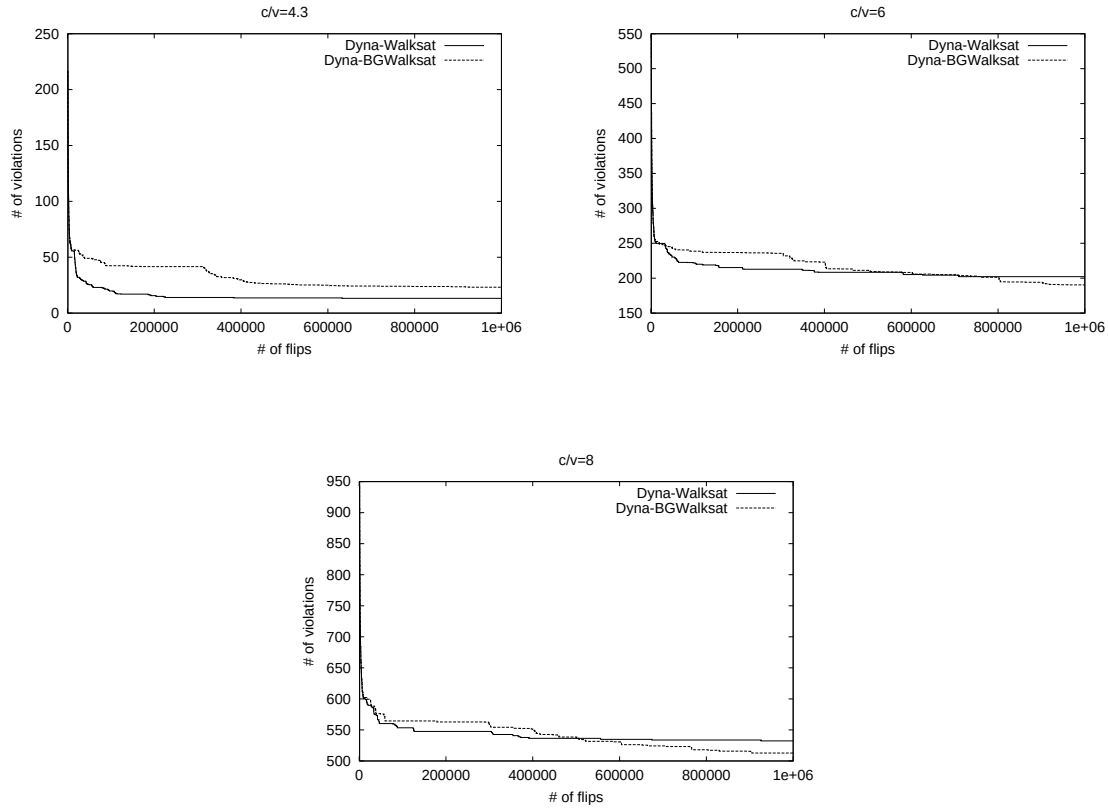


Figure 50: Anytime performance of Dyna-WalkSAT and BG-Dyna-WalkSAT on random Max-3-SAT with 2,000 variables.

information was applied to the search algorithm. This indicates that pseudo backbone information can indeed improve the search performance.

We also investigated the performance of BG-Dyna-WalkSAT with all biased moves as the problem size increases. We considered random Max-3-SAT with C/V ratio fixed at 8.0, and the number of variables from 2,000 to 10,000, with an increment of 2,000. We used 1,000 problem instances for each different size of problems. The results comparing to Dyna-WalkSAT are shown in Table 5.

As the results showed, BG-Dyna-WalkSAT is able to improve upon Dyna-WalkSAT on random Max-3-SAT, especially on overconstrained problem instances.

5.4.2 Problem instances from SATLIB

We compared BG-Dyna-WalkSAT against Dyna-WalkSAT on problem instances from SATLIB [47]. The test problems include SAT-encoded instances from a variety of application domains, including blocks world planning, bounded model checking, all interval series problems, and hard graph coloring problems. We only chose problems with more than 350 variables, and discarded those that can be easily solved by WalkSAT and BGWalkSAT. The chosen problem instances are difficult to solve in general, and their details

# var	*Dyna-WalkSAT*	*BG-Dyna-WalkSAT*	*Diff*
2,000	528.32	497.61	30.71
4,000	1097.60	1041.92	55.68
6,000	1675.80	1597.36	78.44
8,000	2248.19	2159.61	88.58
10,000	2831.32	2724.14	107.18

Table 5: Comparison of BG-Dyna-WalkSAT and Dyna-WalkSAT on random Max-3-SAT with C/V ratio of 8.0, averaged over 1,000 instances. *Diff* is the improvement of BG-Dyna-WalkSAT over Dyna-WalkSAT.

Table 6: BG-Dyna-WalkSAT versus Dyna-WalkSAT on relatively easy satisfiable problems. *Dyna-WalkSAT* and *BG-Dyna-WalkSAT* are the numbers of runs resulting in satisfying solutions (out of 20) by these algorithms. The better results from the two algorithms are underlined and in bold.

problem	*#Var*	*#Clause*	*Dyna-WalkSAT*	*BG-Dyna-WalkSAT*
bw_large.c	3016	50457	1	**2**
bw_large.d	6325	131973	**1**	0
par8-1	350	1149	6	**19**
par8-2	350	1157	6	**19**
par8-3	350	1171	7	**17**
par8-4	350	1155	0	**16**
par8-5	350	1171	1	**15**
qg1-08	512	148957	8	**12**
qg2-08	512	148957	1	**4**
qg3-08	512	10469	11	**20**
qg6-09	729	21844	0	**5**
qg7-09	729	22060	4	**5**
g125.17	2125	66272	**5**	0
g250.29	7250	454622	**4**	2

can be found on the website.

We considered satisfiable and unsatisfiable problems. We ran both Dyna-WalkSAT and BG-Dyna-WalkSAT with a total of ten million flips (compared with one million for our results for random instances) as most of these problem instances are larger than the random Max-3-SAT instances considered in the previous experiments. Interestingly, the average counting (AC) method for computing pseudo backbone frequencies is slightly better than the cost reciprocal averaging counting (CRAC) method. Moreover, biased noise pick and biased clause pick provide substantial improvements to Dyna-WaslkSAT; their combination exhibited superior performance, over a wide range of real instances. In the rest of the section, we present the results of BG-Dyna-WalkSAT using these two biased moves. In our experiments, we ran each of BG-Dyna-WalkSAT and Dyna-WalkSAT twenty times, with each run executing a maximum of ten million flips.

In viewing the results, we found it useful to divide the satisfiable instances into two categories, the easier instances, which were solved at least once (Table 6), and the harder ones, which were not solved by either method, in any of their runs (Table 7). Results for unsatisfiable instances are presented in Table 8.

As the results show, BG-Dyna-WalkSAT significantly outperforms Dyna-WalkSAT in most cases.

Table 7: Dyna-WalkSAT vs. BG-Dyna-WalkSAT on harder satisfiable problems. Dyna-WalkSAT and BG-Dyna-WalkSAT are the average numbers of violations in the best solutions found by the algorithms for a given problem, averaged over 20 runs. Gain is the percentage improvement of BG-Dyna-WalkSAT over Dyna-WalkSAT. The better results are underlined and in bold.

problem	*#Var*	*#Clause*	*Dyna-WalkSAT*	*BG-Dyna-WalkSAT*	*gain (%)*
bmc-ibm-1	9685	55870	25.3	**4.15**	83.60
bmc-ibm-2	3628	14468	5.4	**1.2**	77.78
bmc-ibm-3	14930	72106	115.25	**19.7**	82.91
bmc-ibm-4	28161	139716	118.15	**38.9**	67.08
bmc-ibm-5	9396	41207	12.95	**1.25**	90.35
bmc-ibm-6	51654	368367	358.25	**103.6**	71.08
bmc-ibm-7	8710	39774	17.4	**6.4**	63.22
bmc-galileo-8	58074	294821	65.65	**15.5**	76.39
bmc-galileo-9	63624	326999	95.95	**17.3**	81.97
bmc-ibm-10	61088	334861	406.15	**162.45**	60.00
bmc-ibm-11	32109	150027	439.8	**358.45**	18.50
bmc-ibm-12	39598	19477	554.65	**445.25**	19.72
bmc-ibm-13	13215	6572	88.05	**2.7**	96.93
f2000	2000	8500	2.2	**2.05**	6.82
par16-1-c	317	1264	5.45	**5.35**	1.83
par16-1	1015	3310	10.45	**9.45**	9.57
par16-2-c	349	1392	6.2	**5.9**	4.84
par16-2	1015	3374	10.6	**10.4**	1.89
par16-3-c	334	1332	6	**5.65**	5.83
par16-3	1015	3344	10.45	**9.75**	6.70
par16-4-c	324	1292	6.15	**5.5**	10.57
par16-4	1015	3324	10.4	**9.55**	8.17
par16-5-c	341	1360	6.25	**6.05**	3.20
par16-5	1015	3358	10.45	**9.85**	5.74
par32-1-c	1315	5254	21.7	**20.85**	3.92
par32-1	3176	10277	30.95	**30.25**	2.26
par32-2-c	1303	5206	**21.15**	21.2	-0.24
par32-2	3176	10253	32.1	**28.35**	11.68
par32-3-c	1325	5294	22.05	**21.3**	3.40
par32-3	3176	10297	32.95	**28.55**	13.35
par32-4-c	1333	5326	**21.3**	21.4	-0.47
par32-4	3176	10313	33.65	**29.4**	12.63
par32-5-c	1339	5350	23.15	**22.05**	4.75
par32-5	3176	10325	32.9	**30.3**	7.90
Average					**29.82**

Table 8: Dyna-WalkSAT vs. BG-Dyna-WalkSAT on unsatisfiable problems. The legend is the same as that in Table 7.

problem	*#Var*	*#Clause*	*Dyna-WalkSAT*	*BG-Dyna-WalkSAT*	*gain (%)*
longmult06	2848	8853	**1.5**	1.65	-10.00
longmult07	3319	10335	**2.05**	2.2	-7.32
longmult08	3810	11877	3.65	**2.65**	27.40
longmult09	4321	13479	6.75	**2.9**	57.04
longmult10	4852	15141	10.25	**5.6**	45.37
longmult11	5403	16863	15.05	**9.2**	38.87
longmult12	5974	18645	17.8	**16.2**	8.99
longmult13	6565	20487	23.25	**21.4**	7.96
longmult14	7176	22389	32.6	**24.6**	24.54
longmult15	7807	24351	41.5	**30**	27.71
ssa6288-047	10410	34238	100.25	**89.7**	10.52
Average					**20.01**

On easier satisfiable instances (Table 6), BG-Dyna-WalkSAT finds more satisfying solutions than Dyna-WalkSAT for all parity (par) and quasigroup (qg) classes, and produces similar results to Dyna-WalkSAT on blocksworld (bg) instances. On harder satisfiable instances (Table 7), BG-Dyna-WalkSAT outperforms Dyna-WalkSAT in all but two of 34 instances, where it is less than half a percent worse. In contrast, the overall average gain is about 30%, and the gain is over 50% in 11 of them. On unsatisfiable instances (Table 8), BG-Dyna-WalkSAT produces impressive gains on longmult instances, and on ssa6288-047, with an overall average gain of 20%. On unsatisfiable quasigroup instances (not shown), BG-Dyna-WalkSAT's performance was similar to that of Dyna-WalkSAT. The performance of BG-Dyna-WalkSAT is never more than 10% worse than Dyna-WalkSAT on any of the unsatisfiable instances we studied.

The most glaring failure of BG-Dyna-WalkSAT is on the satisfiable instances $g125.17$ and $g250.29$, shown in Table 6. These instances are SAT-encoded graph coloring problems, and serve to illustrate an important point. As described in Section 5.2, we believe that our method is effective because it exploits the "big valley" structure of the solution space. However, graph coloring problems exhibit a particular type of symmetry in their solution structures which is opaque to local search methods such as WalkSAT. For example, given a solution to a graph coloring problem, swapping red with green results in another solution, which is symmetrical to the original. Thus, there is not a single "big valley" but several in the configuration landscape of the problem, which can bury the true backbone information and thus lead to degraded performance. Presumably, BG-Dyna-WalkSAT's performance will suffer on all instances with this type of symmetry.

5.5 Conclusions and Discussions

In this section, we first carried out a systematic investigation of configuration landscapes of local minima reached by the WalkSAT local search algorithm on random 3-SAT and Max-3-SAT problems. In this analysis, we introduced configuration landscapes to capture the distributions of local minima in terms of

their cost and structural differences. Our analysis revealed that the configuration landscape of a set of local minima from WalkSAT exhibit a single bell-shaped surface, showing that the local minima form a single large cluster. Our results also showed that the WalkSAT algorithm is effective on Max-3-SAT, finding high quality local minima that have large portions of variable assignments consistent with optimal solutions.

Based on the configuration landscape analysis, we developed a novel method to exploit backbone information to improve the performance of a local search algorithm, the WalkSAT algorithm in particular. The main ideas of the method are to extract backbone information from local minima and use it directly to fix possible discrepancies between the current assignment and optimal solutions, so as to guide a local search algorithm towards the regions of search space containing high quality as well as optimal solutions. Our experimental results show that the new method can significantly improve the performance of the WalkSAT local search algorithm on most problem instances from SATLIB, including SAT-encoded problem instances from various applications. On these problem instances, our backbone guided WalkSAT algorithm has a higher probability of reaching satisfiable solutions than the original WalkSAT algorithm, and is able to improve its solution quality on Max-SAT problem instances by 20%.

In retrospect, the most important contributions of and lessons learnt from this section are the idea of using backbone information to improve the performance of a local search algorithm and a simple way of capturing backbone information by using local minima from a local search algorithm. These ideas are general and applicable to other combinatorial problems and other search methods. For example, we have successfully applied the ideas of backbone guided local search to the Traveling Salesman Problem and the Lin-Kernighan local search algorithm [65], which is one of the oldest and most efficient algorithm for the problem [114]. By using structural information such as backbones, the new method drives a search algorithm towards the areas of the search space where most optimal or near optimal solutions are located. In comparison, most existing search techniques focus on the costs of the states in a search space. Therefore, the new algorithm is focused more on where the problems are in the current state, and tries to fix them directly.

One possible drawback of our method is that it requires a good estimation of backbone information. If this estimation deviates substantially from the real backbone information, the new method will not be effective. Nevertheless, the cost reciprocal method for estimate backbone frequencies provides a simple mechanism to ease this problem to some extent by discounting the contribution of a poor local minimum to the pseudo backbone frequencies. Furthermore, most local search methods are randomized algorithms, so better solutions may occasionally be discovered and added to the pool of local minima. Such better local minima will subsequently improve the quality of the estimation of backbone information.

The new backbone guided local search method seems to be not very effective on Max-SAT problems with little structure. The method is particularly hindered by symmetry embedded in a problem. An example of such a problem is graph coloring, where swapping two colors in a solution leads to another solution. In short, backbone guided local search seems to be effective on problems from which significant structural information can be extracted. How to extend the ideas and algorithm presented in this section to address symmetries is an interesting future research topic.

6 An Improved Integer Local Search for Complex Scheduling Problems

The recent advances in the research of Boolean satisfiability (SAT) have provided great insights into the problem, such as phase transitions and backbones [77, 78, 109] (also discussed in Section 4), and have developed efficient algorithms for solving SAT, represented by the widely applied WalkSAT local search algorithm [71, 87] and its variants [45, 71]. The success of WalkSAT has also led to the paradigm of formulating and solving complex planning and scheduling problems as SAT problems [57, 59, 58]. Under this paradigm, a complex problem is encoded as a SAT problem, a solution to the SAT problem is found by applying an algorithm for SAT, and finally the solution is mapped back to the original problem. This SAT-based paradigm has been shown successful for some complex problems in real applications. For example, Blackbox is one of the most competitive methods for planning [58, 59], which was developed under the SAT-based paradigm by applying SAT encoding and SAT algorithms.

Many constraints in real-world applications, however, are complex and may not be easily and efficiently encoded as clauses [15, 28]. More useful and general constraint formulations are integer linear programs (ILP) [41, 98], which allow integer variables and complex constraints, and subsume pseudo Boolean formulae with variables taking values 0 or 1 [39, 97, 98]. ILPs and pseudo Boolean formulae have been extensively applied to planning and scheduling problems [60, 98].

WSAT(oip) [98] is an extension to the WalkSAT algorithm for handling overconstrained integer programs (OIPs) that involve hard and soft constraints. Here a hard constraint is one that needs to be satisfied, and a soft constraint is one that may be violated but incurs a penalty if not satisfied. The objective of such a problem is to satisfy all hard constraints, if possible, or as many hard constraints as possible when overconstrained, while minimizing a penalty function. WSAT(oip) has been shown effective on large constraint optimization and scheduling problems [60, 97, 98].

Inherited from the WalkSAT algorithm, WSAT(oip) is a local search algorithm that makes stochastic local perturbations to the current assignment of all variables in searching for progressively better solutions [98]. A noticeable characteristic of WalkSAT and WSAT(oip) is that whenever multiple choices exist, a *uniformly random* or *unbiased* choice will be made. For example, when an unsatisfied clause is to be selected from a set of unsatisfied clauses, each qualified candidate is given equal change to be picked. Likewise, the variable whose value is to be changed next is chosen, uniformly randomly, from multiple candidates. Such uniform random moves are ineffective when there exist large "plateau" regions in search space, and the problem is exacerbated in OIPs and WSAT(oip) when "plateau" regions become larger due to larger domains of integer variables.

Motivated to solve complex, real-world scheduling problems with hard and soft constraints, we aim to improve WSAT(oip). We particularly introduce three techniques to the existing algorithm. The first is a method of making biased moves in attempting to fix possible discrepancies between the current variable assignment and an optimal solution, so as to drive the search to the regions in search space where high-quality and optimal solutions locate. These biased moves are devised based on our previous work of backbone guided local search for (maximum) Boolean satisfiability [115]. The second method is a sampling-based aspiration search in order to restrict the search to finding progressively improving solutions, so as to reduce

search complexity and increase anytime performance of the resulting algorithm. Our experimental analysis show that this method is particularly effective on problems with hard and soft constraints. The third method is an extension of Hoos's dynamic noise strategy for WalkSAT [46] to WSAT(oip), so that the critical parameter of noise ratio of WSAT(oip) does not have to be tuned for each individual problem instance. The resulting WSAT(oip) becomes more robust, general and flexible for different applications.

This section is organized as follows. We first describe in Section 6.1 our motivating scheduling problem and consider its complexity. We then discuss pseudo Boolean encoding and overconstrained integer programs in Section 6.2, and briefly describe WalkSAT and WSAT(oip) in Section 6.3. We then present the three improving techniques for WSAT(oip) in Section 6.4. We experimentally evaluate the extended WSAT(oip) in Section 6.5, using our scheduling problems and the instances of two scheduling problems from CSPLIB [36]. We finally conclude in Section 6.6.

6.1 Scheduling and Resource Allocation

The specific, motivating scheduling problem of this research is to schedule a large number of training activities for a crew over a period of time, ranging from a few days to a few weeks or months [15, 28]. In such a problem, a trainee needs to finish a set of required activities that requires many trainers and various equipment. These activities are associated with one another by precedent relationships, i.e., one training activity cannot be scheduled until a trainee has finished certain prerequisites. A used equipment (resources) can be reused after some maintenance, which itself is an activity to be scheduled. In addition, individual activities have different importance and carry different penalties if not scheduled. The objective is to schedule as many activities as possible for all the trainees within a gross period of time using the available trainers and equipment so that the penalty of unscheduled activities is minimized. Even though this scheduling problem is not overarchingly sophisticated, it can indeed be viewed as a representative of general scheduling problems with various constraints and being required to optimize an objective function.

At the center of our training scheduling problem, as well as many other similar problems, is a resource allocation problem, i.e., a problem of assigning resources (e.g., trainers and equipment in our scheduling problem) to needy activities. The properties of such an underlying resource allocation problem can help characterize the scheduling problem. The complexity of the former will dominate the complexity of the latter. If the resource allocation problem is difficult, the scheduling problem is doomed to be hard as well.

We now consider a simple, static resource allocation problem that was abstracted from our training scheduling problem at a particular time. We are given a set of n tasks, $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$, and a set of r resources, $\mathcal{R} = \{R_1, R_2, \cdots, R_r\}$. Each task requires a certain number of resources in order to execute, which we call resource requirements. Each resource can only be allocate to one resource requirement, and a resource requirement can be met by having one desirable resource allocated to it. We denote the q_i resource requirements of task T_i by $\mathcal{Q}_i = \{Q_{i,1}, Q_{i,2}, \cdots, Q_{i,q_i}\}$. Table 9 shows a small example of resource requirements of two tasks over three resources. An entity of 1 (0) in the table means that a resource can (cannot) be allocated to the corresponding requirement. In general, the available resources may not be sufficient to fulfill every task; and a task carries a penalty, called *task penalty*, if not scheduled.

		R_1	R_2	R_3
T_1	$Q_{1,1}$	0	1	1
	$Q_{1,2}$	1	0	0
T_2	$Q_{2,1}$	1	1	0
	$Q_{2,2}$	1	1	0

Table 9: A simple resource allocation problem.

The resource allocation problem is to allocate the resources to the tasks so that the overall penalty of unfulfilled tasks is minimized, which constitutes an optimization problem. If all tasks have equal penalties, it is equivalent to fulfill the maximal number of tasks.

Compared to some other resource allocation problems, for instances the permutation problems considered in [91, 99], our problem has a unique, small structure embedded within a task. A task can be scheduled if and only if all its resource requirements are met. We call this feature *bundled resource requirement*. Furthermore, a pair of resource requirements have an exclusive resource contention in that a resource acquired by one requirement cannot be allocated to the others. We call this feature *exclusive resource contention*. To be convenient, we call the problem *bundled, exclusive resource allocation problem*, or *BERAP* for short.

We now show that BERAP is NP-hard [32]. To this end, we prove that a decision version of the problem is NP-complete [32]. A simple, special decision version of BERAP is the following. Given a set of tasks, each of which has a set of resource requirements, decide if at least k tasks can be fulfilled. Here we simply consider every task having a penalty one if it is not fulfilled.

Theorem 6.1 *BERAP with more than two resource requirements per task is NP-complete.*

Proof: We show the above decision version of BERAP is NP-complete. We reduce a NP-complete set packing problem [32] to this decision problem. Given a collection S of finite sets and a positive integer $K \leq |S|$, set packing is to decide if S contains at least K mutually disjoint subsets. Formally, it is to decide if there exists $S' \subseteq S$ such that $|S'| \geq K$ and for all $S_1 \in S'$ and $S_2 \in S'$, $S_1 \cap S_2 = \emptyset$. The problem is NP-complete when every subset $S_i \in S$ has more than two elements. We now reduce an NP-complete set packing problem to our decision BERAP. We map all the elements in the subsets of a set packing problem instance to the resources of BERAP, each subset of the set packing instance to a task of BERAP, and an element in the subset to a resource requirement of the respective task. In other words, the total number of tasks is the number of subsets $|S|$, the number of resources is the number of distinct elements in all subsets of S, and the number of resource requirements of a task is the number of elements in the corresponding subset. Given $K \leq |S|$, the constructed BERAP is to decide if at least K tasks can be fulfilled. Clearly, a solution to the BERAP is also a solution to the original set packing problem. □

This NP-completeness result leads to the conclusion that our crew scheduling problem in intractable in the worst case.

6.2 PB Encoding and Integer Programs

A clause of Boolean variables can be formulated as a linear pseudo Boolean (PB) constraint [39, 98], which we illustrate by an example. We start by viewing Boolean value True (T) as integer 1, and value False (F) as 0. We then map a Boolean variable v to an integer variable x that takes value 1 or 0, and map $\overline{v}$ to $1 - x$. Therefore, when $v = T$, we have $x = 1$ and $1 - x = 0$ which corresponds to $\overline{v} = F$. With this mapping, we can formulate a clause in a linear inequality. For example, $(v_1 \vee \overline{v}_2 \vee v_3)$ can be mapped to $x_1 + (1 - x_2) + x_3 \geq 1$. Here, the inequality means that the clause must be satisfied in order for the left side of the inequality to have a value no less than one. In general, the class of linear PB constraints is defined as $\sum_i c_i \cdot L_i \sim d$, where c_i and d are rational numbers, $\sim$ belongs to $\{=, \leq, <, \geq, >\}$, and the L_i are literals.

However, a clause in an overconstrained problem may not be satisfied so that its corresponding inequality may be violated. To represent this possibility, we introduce an auxiliary integer variable w to the left side of a mapped inequality. Variable $w = 1$ if the corresponding clause is unsatisfied, making the inequality valid; $w = 0$ otherwise. Since the objective is to minimize the number of violated clauses, it is then to minimize the number of auxiliary variables that are forced to take value 1. To be concrete, $(v_1 \vee \overline{v}_2 \vee v_3), (v_2 \vee \overline{v}_4)$ can be written as an overconstrained PB formula of minimizing $W = C_1 \cdot w_1 + C_2 \cdot w_2$, subject to

$$\begin{cases} x_1 & +(1-x_2) & +x_3 & & & +w_1 & \geq 1 \\ & & x_2 & & +(1-x_4) & +w_2 & \geq 1 \end{cases}$$

where C_1 and C_2 are the penalties of the first and second clauses, respectively.

More complex constraint problems, where variables takes integers rather than Boolean values, can be formulated as overconstrained integer programs (OIPs) [98], which are integer linear programs (ILPs) [41] in the sense that they both use inequalities to define the feasible regions of a solution space and aim to optimize an objective function. OIPs differ from ILPs in that OIPs introduce additional, competing soft constraints to encode the overall optimization objective.

A constraint in OIP defines a feasible region for all assignments of the (integer) variables involved. For an assignment that violating a constraint, we can define the distance of the assignment to the boundary of the feasible region specified by the constraint. Such a distance can be measured by the Manhattan distance, the minimum integer distance in the grid space defined by the variable domains. This Manhattan distance was call *score* of the constraint under the given assignment [98]. Obviously, if an assignment satisfies a constraint, then its distance to the constraint boundary is zero.

6.3 The WalkSAT and WSAT(oip) Algorithms

WSAT(oip) belongs to the family of WalkSAT-based local search algorithms, each of which follows the same basic procedure of the WalkSAT algorithm [71, 87]. The WalkSAT algorithm was discussed in Section 5.1.2 and in Figure 45.

WSAT(oip) was built to solve OIPs by extending the WalkSAT algorithm to support integer variables and generic constraints such as inequalities. The main extensions and modifications made by WSAT(oip) are the

following:

- *Distinguishing hard and soft constraints*: When Choosing a violated constraint C, WSAT(oip) selects a violated hard constraint with probability p_h and a violated soft constraint with probability $1 - p_h$.
- *Restricted neighborhood*: When choosing a variable whose value to be changed from all the variables associated with the selected constraint C, only will the (integer) values that differs from the current value by at most d be considered.
- *Tabu search*: When multiple variable-value pairs exist in the greedy choice which make the same amount of improvement to the objective value, break ties first in favor of the one that has been used the least frequently, and then in favor of the one that has not been used the longest.

Similar to WalkSAT WSAT(oip) still uses a noise ratio, which has to be tuned for every problem instance.

6.4 Improvement and Extensions to WSAT(oip)

WSAT(oip) is not very efficient on large problems. We introduce three extensions to improve its performance.

6.4.1 Backbone-guilded biased moves

One observation on local search for SAT problems is that there exist a large amount of plateau regions in the search space where neighboring states all have the same quality. This observation inspired the development of WalkSAT that "walks" on the plateau, thus the name of WalkSAT, by making random moves in order to navigate through plateau regions and to hopefully find downfall edges. Such random, sometimes aimless, plateau moves are not very effective. Even though the use of a tabu list can help prevent to visit the recently visited states [45], the algorithm may still have to explore a large portion of a plateau area.

The inefficacy of WalkSAT's random moves is exacerbated in WSAT(oip) where non-Boolean variables can have large domains, which lead to larger neighborhoods and thus larger plateau regions. Therefore, it is important to shorten or avoid, if possible, such random moves.

The main ideas

Our main idea to address the inefficacy caused by uniformly random moves in WSAT(oip) is to exploit an extended concept of backbone. The backbone variables of an optimization problem are the ones that have fixed values among all optimal solutions; and these backbone variables are collectively called the backbone of the problem. The size of the backbone, the fraction of backbone variables among all variables, is a measure of the constrainedness of a given problem. The concept of backbone variables can be extended to backbone frequencies. The backbone frequency of a variable-value pair is the frequency that the pair appears in all optimal solutions; and the backbone frequency of a variable is the maximum backbone frequency of its values. Specifically, let x be a variable with domain $D = \{v_1, v_2, \cdots, v_k\}$, and $p(x(v_i))$ be the backbone frequency of x taking v_i, then the backbone frequency of x is $p(x) = \max_{v_i \in D}\{p(x(v_i))\}$. Thus, a backbone variable must have backbone frequency of one. The backbone frequency of a variable

captures the tightness of the constraints that the variable is involved; the higher the frequency, the more constrained the variable is.

We can apply backbone frequencies to modify random moves in WSAT(oip). If, somehow, we knew the backbone frequencies of the variable-value pairs of a problem, we could construct a "smart" search algorithm by using the backbone frequency information as an oracle to guide each step of WSAT(oip). At each step of the algorithm, we can use the backbone frequencies to change the way in which a variable is chosen to focus on fixing the critically constrained variables that are not currently set correctly.

Unfortunately, obtaining the exact backbone frequencies of a problem requires to find all optimal solutions, thus is more difficult than finding just one solution. To address this problem, the second key idea of backbone guided local search is to estimate backbone frequencies using local minima from a local search algorithm. We simply treat local minima as if they were optimal solutions and compute *pseudo backbone frequencies*, which are an estimate of real backbone frequencies. More precisely, we define the pseudo backbone frequency of a variable-value pair as the frequency that the pair appears in all local minima.

The quality of pseudo backbone frequencies depends on the effectiveness and efficiency of the local search algorithm used. High-quality local minima can be obtained by efficient local search algorithms. Even though WSAT(oip) may land on suboptimal solutions with fairly high probabilities, most of the local minima from WSAT(oip) are expect to have large portions of variables set to correct values, so that they contain partial optimal solutions or partial backbone. In this research, we directly adopt WSAT(oip) to collect local minima, and then in return apply the backbone guided search method to the algorithm to improve its performance.

Biased moves in WSAT(oip)

Pseudo backbone frequencies can be incorporated in WSAT(oip) to make "biased" moves. Consider an example of two variables, x_1 and x_2, that appear in a violated constraint and have the same effect under the current assignment, i.e., changing the value to one of them makes the violated constraint satisfied, and both variables have the same break-count or will cause the same number of satisfied constraints unsatisfied if changed. Let B be the set of backbone variables along with their fixed values, $\mathcal{T}$ be the set of local minima from which pseudo backbone frequencies were computed, and v_1 and v_2 are the current values of x_1 and x_2. We will prefer to change x_1 over x_2 if under the current assignment, $P\{(x_1 = v_1) \in B|\mathcal{T}\} < P\{(x_2 = v_2) \in B|\mathcal{T}\}$, which means that under the current assignment, x_1 is less likely to be part of backbone than x_2, given the set of local minima $\mathcal{T}$. Note that $P\{(x = v) \in B|\mathcal{T}\}$ is the pseudo backbone frequency of $x = v$ under the evidence of a set of local minima $\mathcal{T}$.

How can the pseudo backbone frequencies be used to alter the way that WSAT(oip) chooses variables? As discussed in Section 6.3, WSAT(oip) *uniformly* randomly chooses a variable when multiple choices exist. For example, when there are multiple variables with zero break-count, WSAT(oip) chooses one arbitrarily. In backbone guided search, we apply pseudo backbone information to force WSAT(oip) to make random but *biased* choices. If two variables can make a constraint satisfied, the variable having a higher backbone frequency will be chosen. In other words, we modify WSAT(oip)'s random strategies in such a way that a

backbone or critically constrained variable will be chosen more often than a less restricted variable. To this end, we use pseudo backbone frequencies to help make random biased selections.

Specifically, we apply pseudo backbone frequencies to modify the random choices made in WSAT(oip). The first random choice in WSAT(oip) is constraint pick, where a violated constraint is selected if multiple ones exist. We want to pick, with high probabilities, those variables that are part of the backbone or highly constrained in all optimal solutions. Therefore, we choose a constraint with the maximal number of critically constrained variables. We use the pseudo backbone frequencies of variables in an unsatisfied constraint, normalized among the violated constraints involved, to measure the degree of constrainedness of the constraint. We then select an unsatisfied constraint among all violated ones based on their degrees of constrainedness. Specifically, let $\mathcal{C}$ be the set of unsatisfied constraints, and q_c the sum of pseudo backbone frequencies of all the variables in a constraint $C \in \mathcal{C}$. We then select constraint C, with probability $p_c = q_c/Q$, from all unsatisfied constraints in $\mathcal{C}$, where $Q = \sum_{C \in \mathcal{C}} q_c$ is a normalization factor.

WSAT(oip) uses three other random rules to arbitrarily select a variable after an unsatisfied constraint is chosen (see Section 6.3 and Figure 45). The flat pick rule chooses a variable from a set of zero break-count variables, if any; the noise pick rule selects one from all variables involved in the chosen constraint; and the greedy pick rule takes a variable among the ones of least break-count. In essence, these rules use the same operation, i.e., picking a variable equally likely from a set of variables. Therefore, we can modify these rules all in the same way by using pseudo backbone frequencies. Let $\{x_1, x_2, \cdots, x_w\}$ be a set of w variables from which one must be chosen, $\{v_1, v_2, \cdots, v_w\}$ their best satisfying assignments (the ones satisfying the constraint and having the highest pseudo backbone frequencies), and $\{p_1, p_2, \cdots, p_w\}$ the pseudo backbone frequencies of variable-value pairs $\{(x_1 = v_1), (x_2 = v_2), \cdots, (x_w = v_w)\}$. Then we choose x_i with probability $p_i / \sum_{j=1}^{w} p_j$.

Furthermore, the idea of pseudo backbone frequencies can also be applied to generate an initial assignment for a local search. Specifically, a variable is assigned a particular value with a probability proportional to the pseudo backbone frequency of the variable-value pair.

The backbone guided WSAT(oip) algorithm

The backbone guided WSAT(oip) algorithm has two phases. The first is the *estimation* phase that collects local minima by running WSAT(oip), with a fixed number of tries. The local minima thus collected are compiled to compute the pseudo backbone frequencies of all variable-value pairs.

The second phase carries out the actual *backbone guided* search, which uses pseudo backbone frequencies to modify the way that WSAT(oip) chooses variables to change. This phase also runs many tries, each of which produces a (new) local minimum. The newly discovered local minima are subsequently added to the pool of all local minima found so far to update the pseudo backbone frequencies.

6.4.2 Aspiration search

Solving an OIP requires to optimize two (conflicting) objectives, satisfying the maximum number of hard constraints and minimizing a penalty function of soft constraints violated. Two obvious methods can be taken to make a balance between these two objectives. One is to directly search for a solution by

considering hard and soft constraint together, which was suggested and taken in WSAT(oip) [98], which attempts to select a variable involved in a hard constraint with probability p_h and a variable associated with a soft constraint with probability $1 - p_h$. Note that the performance of WSAT(oip) depends to a large degree on this parameter. The other way is to satisfy the maximum number of hard constraints first and then try to minimize the total penalty of violated soft constraints. However, these two methods do not work very well on large, complex OIPs.

In many real-world constraint problems, our training scheduling problems discussed in Section 6.1 in particular, the number of hard constraints may be large; finding the best assignment to the variables involved in hard constraints may itself be a costly task. Even if such an optimal solution can be found, it may not be extended to an overall assignment of minimal penalty. Therefore, many optimal assignments to the variables in hard constraints must be examined, making the overall search prohibitively costly.

To make WSAT(oip) efficient on OIPs, we propose what we call *aspiration search* strategy. An aspiration level corresponds to a targeted penalty score; the higher an aspiration level, the lower the targeted penalty score. Given an aspiration level, we first search for an assignment so that the total penalty of unsatisfied soft constraints is no more than the targeted penalty value. When such an assignment is found, we attempt to extend the current partial assignment to satisfy the maximum number of hard constraints. This process of extending a partial assignment to a complete assignment may be repeated many times; and the maximum number of hard constraints satisfied S_h is recorded. Each such process corresponds to a probing in the search space under the current aspiration level. We then increase the aspiration level and repeat the processes of probing with the objective of finding an assignment that violates no more less than S_h hard constraints and whose penalty meets the restriction of the current new aspiration level. S_h is also updated if a better assignment, one violating less hard constraints, is found under the current aspiration level. This means that the overall processes attempt to find progressively better solutions for both hard and soft constraints. If we fail to find an assignment satisfying at least S_h hard constraints and keeping the penalty above the current aspiration level after a certain number of tries, the algorithm terminates and the best solution found so far is returned.

Aspiration search has several advantages. First, it decomposes an OIP into several decision problems, each of which has a different degree of constrainedness represented by an aspiration level. At a given aspiration level, this strategy also integrates a sampling method, which first probes the search space to reach a partial assignment such that the penalty function is above the aspiration level, with a search for satisfaction of the hard constraints. Thanks to the partial assignment, the hard constraints can be simplified, as the variables involved in soft constraints are instantiated, so that optimizing hard constraints becomes relatively easier. As a result, aspiration search is able to reduce search cost. Second, the probability p_h of choosing a variable involved in hard constraints, an important parameter determining the performance of WSAT(oip), disappears, making the algorithm less problem dependent. Third, the aspiration search strategy can interact closely with the backbone-guided local search method, making the latter more effective. Since aspiration search is able to reach progressively better solutions, the suboptimal solutions at various aspiration levels can thus be used as local minima to compute pseudo backbone frequencies, which expedites

the process of gathering backbone frequency information.

When applying sampling method, it is desirable, albeit difficult, to know if the current partial assignment at a certain aspiration level can be extended to satisfy at least S_h hard constraints. Our approach to this problem is to monitor the progress of extending the partial assignment to a full assignment. If the number of violated hard constraints decreases after a fixed number of moves M, we consider the current partial assignment extensible. Otherwise, the current partial assignment will be abandoned, and another partial assignment above the current aspiration level will be sampled. Note that the performance of the overall search is affected by the fixed number of moves M within which a better complete assignment must be found. If this number is too large, we may waste too much time on an unsatisfiable deadend; whereas if it is too small, we may miss a satisfiable sample. We develop a dynamic method to adjust this parameter M in our extended WSAT(oip) algorithm. The detail of this method will be discussed in the next subsection where we collectively deal with the issues of how to dynamically adjust the parameters of the WSAT(oip) and extended WSAT(oip) algorithm.

6.4.3 Dynamic, adaptive parameters

One limitation of the WalkSAT family of algorithms, including WSAT(oip), is its dependence on a manually set noise ratio, which is the probability of how often a nongreedy move should be taken (see Section 6.3). In addition, whenever a noise ratio is chosen it will be used throughout the search. It is evident that big progresses can be more easily made at an early stage of a local search than at a late stage. Therefore the noise ratio should be adjusted dynamically depending on where the current search is in the overall search space.

The dynamic noise strategy proposed in [46] for WalkSAT is one such method. The idea of this strategy is simple: start a local search with the noise ratio equal to zero, and examine the number of violations in the current state every θm flips, where m is the number of constraints of a given problem, and θ a constant. If the number of violations has not decreased since the last time we checked (θm flips ago), the search is assumed to have stagnated, and the noise ratio is increased to $wp + (1 - wp)\phi$, where wp is the current noise ratio and ϕ is another constant. Otherwise, the noise ratio is decreased to $wp(1 - 2\phi)$. The discrepancy between the formulas for increasing and decreasing the noise ratio is based on some empirical observations of how WalkSAT behaves when the noise ratio is too high, compared with how it behaves when the parameter is too low [46]. We refer to this strategy as Dyna-WalkSAT for convenience.

Dyna-WalkSAT uses two parameters, $\theta$and ϕ. The difference of using these two new parameters and using the noise ratio in the original algorithm is that these two parameters do not have to be tuned for every single problem instance; the performance of Dyna-WalkSAT with the same values for $\theta$and ϕ is relatively consistent across different problem instances.

Although Dyna-WalkSAT was originally designed and tested on WalkSAT for SAT, we have found it effective on WSAT(oip) for pseudo Boolean encoded problems and OIPs. We call WSAT(oip) using the dynamic noise strategy Dyna-WSAT(oip). Following [46] we set $\theta = 1/6$ and $\phi = 1/5$ in Dyna-WSAT(oip), which have been found to be effective over a wide range of problem instances. Due to its simplicity and

Problem		WSAT(oip)			EWSAT(oip)		
n	m	unsat	penalty	time	unsat	penalty	time
11520	16308	12.43	10380	287.4	**12.08**	**9330**	**167.6**
21240	34908	10.58	8550	824.7	**8.05**	**6270**	**490.2**
21800	37501	3.45	2520	**56.0**	**3.35**	**2325**	56.3
40718	72071	3.95	2970	**26.2**	**3.88**	**2880**	41.2
41496	66892	11.93	10230	1936.4	**8.70**	**6810**	**965.2**
79580	143896	4.10	3120	514.5	**3.97**	**2963**	**344.3**

Table 10: Comparison on crew training scheduling, where n and m are the numbers of variables and clauses, respectively; *unsat* is the average number of violated hard constraints, *penalty* the average penalty score, and *time* the average CPU time in seconds. The better results between the two algorithms are in bold.

reasonable performance, in the rest of this subsection we will use Dyna-WSAT(oip) with $\theta = 1/6$ and $\phi = 1/5$ as default parameters in our experimental analysis.

As mentioned at the end of the previous section, another parameter in our extended WSAT(oip) algorithm is the number of moves M between two consecutive check points for examining progress, if any, made by the algorithm in extending the current partial assignment to a complete one that satisfies the maximum number of hard constraints under the current aspiration level. A good value for parameter M can be achieved when a good balance is made between the algorithm's ability to find satisfied solutions and its ability to escape from local minima. This leads to our adaptive parameter approach, in which the parameter M is dynamically adjusted based on progress made or not made, as reflected in the time elapsed since the last improvement made to hard constraints. At the beginning of the search, we give an initial value that is proportional to the number of hard constraints m to parameter M. If the number of hard constraints does not decrease over the last M search steps, we increase M to $M + k_1 m$, where k_1 is a positive constant less than one. Otherwise, we decrease M by $k_2 m$, where k_2 is another positive constant less than one. In our experiments, we took $M = \min\{m, K/10\}$ and $k_1 = k_2 = M/5$, where K is the maximum number of moves for a restart.

6.5 Applications and Experimental Evaluation

We implemented an improved and extended WSAT(oip) algorithm that incorporates the backbone-guided biased moves, sampling-based aspiration search and dynamic parameter strategies. We shorthand the improved WSAT(oip) as EWSAT(oip). In this section, we report the experimental results, comparing the EWSAT(oip) algorithm with its predecessor, the WSAT(oip) algorithm. We carried out our analyses on three different scheduling problems: our crew training scheduling problem (Section 6.1), progressive party scheduling and basketball tournament scheduling. The last two problems were studied in [98], and also included as benchmark problems in CSPLIB [36], an online repository of CSP problems. All our experiments were run on an AMD Athelon 1900 machine with 2GB memory.

In our experiments, we tried various parameter settings for the WSAT(oip) algorithm, which include the probability p_h of choosing a hard violated constraint over a soft violated constraint and the size of

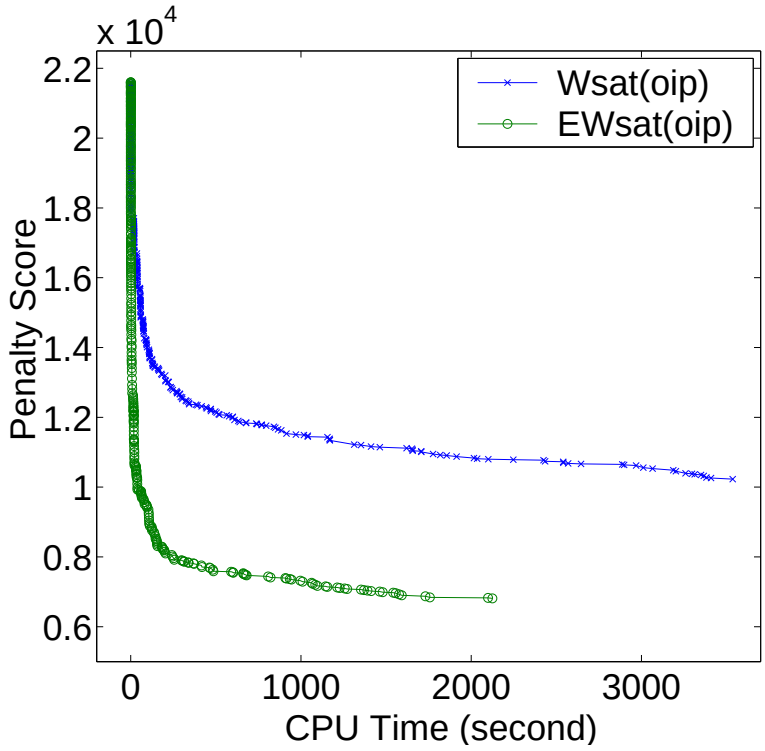


Figure 51: Anytime comparison on a crew scheduling.

tabu list. The comparison results below are for the best parameters for WSAT(oip). Specifically, on our crew scheduling problems, the best probability p_h is 99%, and on the party scheduling and basketball scheduling problems, the best p_h is 90%. To make a fair comparison, we applied dynamic parameter method to automatically adjust the noise ratios for both WSAT(oip) and EWSAT(oip), and let EWSAT(oip) have the same size of tabu list as used by WSAT(oip), which was set to 4.

6.5.1 Crew training scheduling

The first and main problem we considered is our crew training scheduling problem. The test set consists of six large and many small problem instances, derived from a real application domain involved with a large number of crew members of different specialty and various equipment that requires routine maintenance. These problem instances vary in sizes and degree of constrainedness; the largest instances has 79,580 variables and 143,896 constraints. These problems were collected from overconstrained situations and their hard constraints did not seem to be satisfiable all together. Here we present the results on these six large instances.

In our experiments, we allowed WSAT(oip) and EWSAT(oip) to have 200 random restarts for each of their run, and 60,000 maximum moves (variable-value changes) with each restart. The average results comparing these two algorithms over 40 runs on all six problem instances are shown in Table 10. We examined the average minimum number of unsatisfied hard constraints (unsat), the average minimum penalties (penalty) and the average CPU time (time) required to reach solutions of such qualities. As the

Problem			WSAT(oip)		EWSAT(oip)	
name	n	m	median	average	median	average
ppp:1-12,16	4662	31725	**0.175**	**0.185**	0.186	0.190
ppp:1-13	4632	30964	0.406	0.441	**0.388**	**0.423**
ppp:1,3-13,19	4608	30348	0.469	0.472	**0.388**	**0.449**
Ppp:3-13,25,25	4644	31254	**0.625**	**0.713**	0.656	0.718
ppp:1-11,19,21	4602	30179	15.283	15.814	**9.453**	**12.856**
ppp:1-9,16-19	4626	30747	44.906	63.553	**34.546**	**58.302**

Table 11: Comparison on progressive party scheduling problem, where n and m are the numbers of variables and clauses, respectively, and *median* and *average* are the median and average CPU times in seconds. The better results are in bold.

results show, EWSAT(oip) is able to find better solutions with more hard constraints satisfied and lower penalties for all problem instances, and some times with less execution time.

Additional insights were gained when we examined the anytime performance of the two algorithms on these difficult problems. Figure 51 shows such an anytime comparison on one of the six instances, also averaged over 40 runs. As shown, EWSAT(oip) can make significant improvement in an early stage of the search, indicating that it explores more fruitful regions of the search space more effectively than WSAT(oip). The anytime results on the other problem instances were similar to that in Figure 51.

6.5.2 Progressive party scheduling

The Progressive Party Problem (PPP) is to progressively timetable a sequence of parties. There are a total of six critically constrained PPP problem instances in CSPLIB [36], which are all satisfiable. We considered all of them in our experiments. We allowed 100 random restarts in one run of the algorithm, each of which used 60,000 moves. We averaged the results over 40 runs for each of two algorithms. The problem sizes are relatively small, comparing to the crew scheduling problems, with the number of variables less than five thousand and the number of constraints no more than 32,000. Table 11 shows the median and average CPU times to reach satisfying solutions by WSAT(oip) and EWSAT(oip).

On four of the six problem instances, both algorithms take less than one second to finish and have similar performance. On the other two problem instances, EWSAT(oip) can reduce median execution time from 15.3 seconds to 9.5 seconds and 44.9 seconds to 34.6 seconds, giving time reductions of 37.9% and 22.9%, respectively.

6.5.3 Basketball tournament scheduling

The Atlantic Coast Conference (ACC) Basketball Scheduling Problem is to arrange a basketball tournament in the ACC. The problem was originally described by Michael Trick and George Nemhauser. Walser developed a pseudo Boolean integer linear programming model for these problems [98]. The objective of an encoded ACC scheduling problem is to satisfy all the hard constraints while minimizing the total penalty caused by violated soft constraints. The difficulties of the available problem instances vary dramatically.

Problem			WSAT(oip)		EWSAT(oip)	
name	n	m	median	average	median	average
acc-tight:2	1620	2520	0.86	**1.03**	**0.77**	1.05
acc-tight:3	1620	3249	1.30	2.16	**1.26**	**1.83**
acc-tight:4	1620	3285	44.14	61.49	**35.35**	**48.69**
acc-tight:5	1339	3052	1171.17	1071.18	**603.57**	**609.05**

Table 12: Comparison on ACC backetball scheduling problem, where the legends are the same as in Table 11.

Here we only consider four instances of moderate difficulties.

The experiment setup was the same as for the party scheduling problem considered earlier. The median and average times to reach satisfying solutions to these problems are included in Table 12. EWSAT(oip) outperformed WSAT(oip) on all these instances except the slightly slower average time on instance acc-tight:2. The performance of EWSAT(oip) seems to improve on hard instances. For example, EWSAT(oip) reduced the median running time by 48.4% on acc-tight:5, increased from 20.5% on acc-tight:4.

6.6 Conclusions

WSAT(oip) is an extensively applied integer local search algorithm for solving constraint problems with hard and soft constraints which are represented as overconstrained integer linear programs (OIPs). In this section, we introduced three strategies to improve the performance and applicability of WSAT(oip) in solving complex scheduling problems: biased-move strategy to improve the efficacy of local search by exploiting backbone structures; sampling-based aspiration search to find high quality solutions and improve anytime performance; dynamic parameter adaptation to make WSAT(oip) robust and more applicable to real-world problems. Our experimental results on three large and complex scheduling problems show that our improved WSAT(oip) algorithm significantly improves upon the original WSAT(oip) by finding better solutions on overconstrained problems or finding better or same-quality solutions sooner. We expect that these new methods can be applied to other search algorithms and combinatorial optimization problems.

The most important lesson we have learnt from this research was that single method may not be effective. It usually requires to combine many different strategies in order to solve difficult problem instances with a reasonable amount of time.

References

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable instances. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-02)*, 2000.

[2] Mark R. Adler, Alvah B. Davis, Robert Weihmayer, and Ralph W. Worrest. Conflict-resolution strategies for nonhierarchical distributed agents. In *Distributed Artifical Intelligence*, volume 2, pages 139–162. 1989.

[3] D. J. Aldous. The $\zeta(2)$ limit in the random assignment problem. *Random Structures and Algorithms*, 18:381–418, 2001.

[4] E. Balas and P. Toth. Branch and bound methods. In *The Traveling Salesman Problem*, pages 361–401. John Wiley & Sons, Essex, England, 1985.

[5] M. N. Barber. Finite-size scaling. In *Phase Transitions and Critical Phenomena*, volume 8, pages 145–266. Academic Press, 1983.

[6] J. C. Beck and M. S. Fox. A generic framework for constraint-directed search and scheduling. *AI Magazine*, 19(4):101–130, 1998.

[7] M. Bellmore and J. C. Malone. Pathology of traveling-salesman subtour-elimination algorithms. *Operations Research*, 19:278–307, 1971.

[8] K. D. Boese. *Models for Iterative Global Optimization*. PhD thesis, UCLA/Computer Science Department, 1996.

[9] K. D. Boese, A. B. Kahng, and S. Muddu. New adaptive multistart techniques for combinatorial global optimizations. *Operations Research Letters*, 16, 1994.

[10] C. Borgs, J. T. Chayes, and B. Pittel. Phase transition and finite-size scaling for the integer partitioning problem. *Random Structures and Algorithms*, 19:247–288, 2001.

[11] Stephanie Cammarata, David McArthur, and Randall Steeb. Strategies of cooperation in distributed problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 767–770, 1983.

[12] G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large-scale, asymmetric Traveling Salesman Problems. *ACM Trans. on Mathematical Software*, 21:394–409, 1995.

[13] G. Carpaneto and P. Toth. Some new branching and bounding criteria for the asymmetric traveling salesman problem. *Management Science*, 26:736–743, 1980.

[14] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, Sydney, Australia, August 1991.

[15] J. Chen, A. Bugacov, P. Szekely, M. Frank, M. Cai, D. Kim, and R. Neches. Coordinated aggressive bidding in distributed combinatorial resource allocation. In *Proc. AAMAS Workshop on Representations and Approaches for Time-critical Decentralized Resource/Role/Task Allocation*, Melbourne, Australia, July 2003.

[16] P. Codognet and F. Rossi. Notes for the ECAI2000 tutorial on Solving and Programming with Soft Constraints: Theory and Practice. available at http://www.math.unipd.it/ frossi/papers.html.

[17] Susan E. Conry, Robert A. Meyer, and Victor R. Lesser. Multistage negotiation in distributed planning. In *Readings in Distributed Artifical Intelligence*, pages 367–384. 1988.

[18] D. Coppersmith and G. B. Sorkin. Constructive bounds and exact expectations for the random assignment problem. *Random Structures and Algorithms*, 15:113–144, 1999.

[19] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of ACM*, 5:394–397, 1962.

[20] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[21] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of ACM*, 32:505–536, 1985.

[22] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formula. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence, (IJCAI-01)*, pages 248–253, 2001.

[23] Edmund H. Durfee and Victor R. Lesser. Negotiation task decomposition and allocation using partial global planning. In *Distributed Artifical Intelligence*, volume 2, pages 229–244. 1989.

[24] Edmund H. Durfee and Thomas A. Montgomery. A hierarchical protocol for coordinating multiagent behaviors. In *Proceedings of the Eight National Conference of Artificial Intelligence*, pages 86–93, 1990.

[25] M. Fabiunke. Parallel distributed constraint satisfaction. In *Proc. Intern. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA-99)*, pages 1585–1591, 1999.

[26] S. Fitzpatrick and L. Meertens. Experiments on dense graphs with a stochastic, peer-to-peer colorer. In *AAAI-02 Workshop on Probabilistic Approaches in Search, to appear*.

[27] S. Fitzpatrick and L. Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In *Proc. 1st Symp. on Stochastic Algorithms: Foundations and Applications*, pages 49–64, 2001.

[28] M. Frank, A. Bugacov, J. Chen, G. Dakin, P. Szekely, and B. Neches. The marbles manifesto: A definition and comparison of cooperative negotiation schemes for distributed resource allocation. In *Proc. AAAI-01 Fall Symposium on Negotiation Methods for Autonomous Cooperative Systems*, pages 36–45, 2001.

[29] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[30] A. Frieze, R. M. Karp, and B. Reed. When is the assignment bound asymptotically tight for the asymmetric traveling-salesman problem? In *Proc. of Integer Programming and Combinatorial Optimization*, pages 453–461, 1992.

[31] A. Frieze and G. B. Sorkin. The probabilistic relationship between the assignment and asymmetric traveling salesman problems. In *Proc. of SODA-01*, pages 652–660, 2001.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, NY, 1979.

[33] I. Gent and T. Walsh. The TSP phase transition. *Artificial Intelligence*, 88:349–358, 1996.

[34] I.P. Gent and T. Walsh. Phase transitions and annealed theories: Number partitioning as a case study. In *Proceedings of 12th ECAI*, 1996.

[35] I.P. Gent and T. Walsh. Analysis of heuristics for number partitioning. *Computational Intelligence*, 14(3):430–451, 1998.

[36] I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available at http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[37] C. P. Gomes, T. Hogg, T. Walsh, and W. Zhang. IJCAI-2001 tutorial: Phase transitions and structure in combinatorial problems. http://www.cs.wustl.edu/∼zhang/links/ijcai-phase-transitions.html.

[38] G. Gutin and A. P. Punnen, editors. *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers, 2002.

[39] P. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer, 1968.

[40] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro.*, 22(6):12–24, 2002.

[41] F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 7 edition, 2002.

[42] K. Hirayama, M. Yokoo, and K. Sycara. The phase transition in distributed constraint satisfaction problems: First results. In *Proc. Intern. Workshop on Distributed Constraint Satisfaction*, 2000.

[43] T. Hogg. Exploiting problem structure as a search heuristic. Technical report, Xerox PARC, January 1995.

[44] T. Hogg, B. A. Huberman, and C. Williams. Phase transitions and the search problem. *Artificial Intelligence*, 81:1–15, 1996.

[45] H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666, 1999.

[46] H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 655–660, Edmonton, Canada, July 28-Aug. 1 2002.

[47] H. H. Hoos and T. Stuzle. SATLIB - the satisfiability library. http://www.informatik.tu-darmstadt.de/AI/SATLIB, 1999.

[48] B. A. Huberman and T. Hogg. Phase transitions in artificial intelligence systems. *Artificial Intelligence*, 33:155–171, 1987.

[49] E. T. Jaynes. The rationale of maximum entropy methods. *Proc. IEEE*, 70:939–952, 1982.

[50] Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochstic algorithm for MAX-SAT. In *Proc. 1st Workshop on AI and OR*, Timberine, OR, 1995.

[51] D. S. Johnson, G. Gutin, L. A. McGeoch, A. Yeo, W. Zhang, and A. Zverovitch. Experimental analysis of heuristics for the ATSP. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 445–488. Kluwer Academic Publishers, 2002.

[52] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Comlexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[53] R. M. Karp. A patching algorithm for the nonsymmetric traveling-salesman problem. *SIAM Journal on Computing*, 8:561–573, 1979.

[54] R. M. Karp. An upper bound on the expected cost of an optimal assignment. In D. Johnson, editor, *Discrete Algorithms and Complexity: Proc. of the Japan-US Joint Seminar*, pages 1–4, New York, 1987. Academic Press.

[55] R. M. Karp and J. Pearl. Searching for an optimal path in a tree with random costs. *Artificial Intelligence*, 21:99–117, 1983.

[56] R. M. Karp and J. M. Steele. Probabilistic analysis of heuristics. In *The Traveling Salesman Problem*, pages 181–205. John Wiley & Sons, Essex, England, 1985.

[57] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201, 1996.

[58] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, 1998.

[59] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

[60] H. Kautz and J. P. Walser. Integer optimization models of AI planning problems. *Knowledge Engineering Review*, 15:101–117, 2000.

[61] S. Kirkpatrick and G. Toulouse. Configuration space analysis of traveling salesman problems. *J. Phys. (France)*, 46:1277–1292, 1985.

[62] Susan E. Lander and Victor R. Lesser. Customizing distributed search among agents with heterogeneous knowledge. In *Proceedings of the First International Conference on Information and Knowledge Management*, pages 335–344, 1992.

[63] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, Essex, England, 1985.

[64] Victor R. Lesser. An overview of dai: Viewing distributed ai as distributed search. *Journal of Janpanese Society for Artificial Intelligence-Special Issue on Distributed Artificial Intelligence*, 5(4):392–400, January 1990.

[65] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the Traveling Salesman Problem. *Operations Research*, 21:498–516, 1973.

[66] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[67] W. G. Macready, A. G. Siapas, and S. A. Kauffman. Criticality and parallelism in combinatorial optimization. *Science*, 271, 1996.

[68] Roger Mailler, Regis Vincent, Victor lesser, Jiaying Shen, and Tim Middlekoop. Soft-real time, cooperative negotiation for distributed resource allocation. In *AAAI Fall Symposium on Negotiation Methods for Autonomous Cooperative Systems*, 2001.

[69] S. Martello and P. Toth. Linear assignment problems. *Annals of Discrete Mathematics*, 31:259–282, 1987.

[70] O. C. Martin, R. Monasson, and R. Zecchina. Statistical mechanics methods and phase transitions in optimization problems. *Theoretical Computer Science*, 265:3–67, 2001.

[71] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 321–326, 1997.

[72] C. J. H. McDiarmid. Probabilistic analysis of tree search. In G. R. Gummett and D. J. A. Welsh, editors, *Disorder in Physical Systems*, pages 249–260. Oxford Science, 1990.

[73] C. J. H. McDiarmid and G. M. A. Provan. An expected-cost analysis of backtracking and non-backtracking algorithms. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 172–177, Sydney, Australia, August 1991.

[74] M. Mézard and G. Parsi. On the solution of the random link matching problem. *J. Physique*, 48:1451–1459, 1987.

[75] M. Mézard, G. Parsi, and M. A. Virasoro, editors. *Spin Glass Theory and Beyond*. World Scientific, Singapore, 1987.

[76] Steven Minton, Andy Philips, Mark D. Johnston, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[77] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, San Jose, CA, July 1992.

[78] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.

[79] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 40–45, Washington, DC, July 1993.

[80] C. H. Papadimitriou and M. Yannakakis. The travelling salesman problem with distances one and two. *Math. Oper. Res.*, 18:1–11, 1993.

[81] D. J. Patterson and H. Kautz. Auto-WalkSAT A self-tuning implementation of walkSAT. In *Electronic Notes in Discrete Mathematics (ENDM), 9, 2001, Elsevier. Presented at the LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, Boston University, MA, June 14-15, 2001.

[82] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.

[83] J. Pearl. Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32:245–257, 1987.

[84] J. C. Pemberton and W. Zhang. Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems. *Artificial Intelligence*, 81:297–325, 1996.

[85] H. Reichl. Overview and development trends in the field of MEMS packaging. invited talk given at 14th Intern. Conf. on Micro Electro Mechanical Systems, Jan. 21-25, 2001, Switzerland.

[86] Jr. S. Kirkpatrick, C. D. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[87] B. Selman, H. Kautz, and B. Cohen. Noise strategies for local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343, Seattle, WA, 1994.

[88] B. Selman and S. Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *Artificial Intelligence*, 81:273–295, 1996.

[89] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, San Jose, CA, July 1992.

[90] J. Slaney and T. Walsh. Backbones in optimization and approximation. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 254–259, 2001.

[91] B. M. Smith. Modelling a permutation problem. In *Proc. of ECAI-00 Workshop on Modelling and Solving Problems with Constraints*, 2000.

[92] T. H. C. Smith, V. Srinivasan, and G. L. Thompson. Computational performance of three subtour elimination algorithms for solving asymmetric traveling salesman problems. *Annals of Discrete Mathematics*, 1:495–506, 1977.

[93] M. Takeda. Applications of MEMS to industrial inspection. invited talk, 14th Intern. Conf. on Micro Electro Mechanical Systems, Jan. 21-25, 2001.

[94] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2 edition, 2000.

[95] O. Telelis and P. Stamatopoulos. Heuristic backbone sampling for maximum satisfiability. In *Proc. 2nd Hellenic Conf. on Artificial Intelligence*, pages 129–139, 2002.

[96] D. W. Walkup. On the expected value of a random assignment problem. *SIAM Journal on Computing*, 8:440–442, 1979.

[97] J. P. Walser. Solving linear pseudo-boolean constraint problems with local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[98] J. P. Walser. *Integer Optimization by Local Search*. Springer, 1999.

[99] T. Walsh. Permutation problems and channeling constraints. In *Proc. IJCAI-01 Workshop on Modeling and Solving Problems with Constraints*, pages 125–133, 2001.

[100] G. Wang, W. Zhang, R. Mailler, and V. Lesser. *Distributed Sensor Networks: A Multiagent Systems Approach*, chapter Analysis of negotiation protocols by distributed search, pages 339–362. Kluwer, 2003.

[101] K. G. Wilson. Problems in physics with many scales of length. *Scientific American*, 241:158–179, 1979.

[102] Z. Xing and W. Zhang. A constraint sensitive algorithm for maximum satisfiability. manuscript under preparation, 2003.

[103] M. Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer Verlag, Berlin, Heidelberg, New York, 2001.

[104] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10(5):673–685, 1998.

[105] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS-96)*, pages 401–408, 1996.

[106] Makoto Yokoo. *Distributed Constraint Satisfaction*. Springer, 1998.

[107] W. Zhang. *State-space Search: Algorithms, Complexity, Extensions, and Applications*. Springer, 1999.

[108] W. Zhang. Phase transitions and backbones of 3-SAT and Maximum 3-SAT. In *Proc. 7th Intern. Conf. on Principles and Practice of Constraint Programming (CP-2001)*, pages 153–167, 2001.

[109] W. Zhang. Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *Proc. Intern. Conf. on Principles and Practice of Constraint Programming (CP-01)*, pages 153–167, 2001.

[110] W. Zhang. Phase transitions of the asymmetric traveling salesman. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1202–1207, 2003.

[111] W. Zhang, Z. Deng, G. Wang, L. Wittenburg, and Z. Xing. Distributezhangww-dsa,d problem solving in sensor networks. In *Proc. AAMAS-02*, 2002.

[112] W. Zhang and R. E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79:241–292, 1995.

[113] W. Zhang and R. E. Korf. A study of complexity transitions on the asymmetric Traveling Salesman Problem. *Artificial Intelligence*, 81:223–239, 1996.

[114] W. Zhang and M. Looks. Backbone guided local search for the traveling salesman. In *presented at the 5-th Metaheuristic International Conference*, Kyoto, Japan, August 2003.

[115] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1179–1184, 2003.

[116] W. Zhang, G. Wang, and L. Wittenburg. Distributed stochastic search for distributed constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *Proc. AAAI-02 Workshop on Probabilistic Approaches in Search*, pages 53–59, 2002.

[117] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. *Distributed Sensor Networks: A Multiagent Systems Approach*, chapter A comparative study of distributed constraint algorithms with applications to problems in sensor networks, pages 319–338. Kluwer, 2003.

[118] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 352–357, 2002.

[119] W. Zhang and Z. Xing. Distributed breakout vs. distributed stochastic: A comparative evaluation on scan scheduling. In *Proc. AAMAS-02 Workshop on Distributed Constraint Reasoning*, to appear.

[120] W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *Proc. 2nd Intern. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-03)*, 2003.

[121] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *Proc. 1st IEEE Intern. Workshop on Sensor Network Protocols and Applications Anchorage*.